Polymorphism

Polymorphism in Java; method overloading, overload resolution rules, covariant return types, polymorphism vs inheritance, and the Strategy pattern via composition.

Intermediate 45 min

Generated on 2 October 2026
Read online: https://www.learnforgex.com/en/topics/polymorphism

---

Polymorphism

The Inheritance lesson's "Method Overriding" and "Upcasting" sections already showed us the most common form of polymorphism — runtime polymorphism — without ever naming it: in `Animal animal = new Dog(); animal.makeSound();`, which implementation actually runs was decided at runtime. In this lesson we'll put a formal name on that observation ("polymorphism"), meet its lesser-known sibling — **method overloading**, resolved at compile time — and see how polymorphism combines with interfaces, abstract classes, and composition.

What Is Polymorphism?

Polymorphism comes from the Greek "poly" (many) and "morph" (form) — it means the same interface (the same method call) can produce different behavior depending on the actual object type: **"one interface, many implementations."** Java has two distinct flavors of this:

- **Compile-time polymorphism:** The compiler decides which method runs by looking at the arguments in the call. This is method overloading.
- **Runtime polymorphism:** Which implementation runs is decided at runtime, based on the object's actual type. This is method overriding.

```
animal.makeSound(); // which makeSound() runs? -- depends on the object's real type
(runtime)
print("hello");     // which print() runs? -- depends on the argument's type (compile-time)
```

Why Does It Exist?

A real-world example: imagine a payment system with several methods — `CreditCard`, `PayPal`, `BankTransfer`. Without polymorphism you'd have to check every payment method

with its own `if/else`: `if (type.equals("CARD")) {...} else if (type.equals("PAYPAL")) {...}`. Every time a new payment method is added, you'd have to find and update **every** spot in that chain — a fragile design that only gets harder to manage as it grows. Polymorphism lets each payment method implement a shared interface (`process()`) its own way, so calling code runs with a single line (`payment.process()`) and never needs to know which payment method it's actually dealing with. We'll build this idea end-to-end in the first mini project.

History

Like inheritance, the concept of polymorphism traces back to Simula 67 in 1967 — but the two forms Java inherited have different roots: method overriding (runtime polymorphism) comes directly from Simula/Smalltalk's object-oriented lineage, while method overloading (compile-time polymorphism) is a much older idea, present even in procedural languages (like Ada) — "multiple functions with the same name but different signatures." Java has supported both since its first release in 1996, and the compiler draws a sharp line between when each applies: overloading is always resolved at compile time, based on the signature; overriding is always resolved at runtime, based on the object's actual type. That clarity is a deliberate departure from the "duck typing" approach dynamically-typed languages (like Python or Ruby) take — in Java, which methods an object can be called with is determined at compile time by the static type, not at runtime.

Compile-Time vs. Runtime Polymorphism

Seeing these two forms of polymorphism side by side makes the difference clear:

```
class Animal {
    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }
}

class PolymorphismOverviewExample {
    // Compile-time polymorphism: the compiler picks ONE of these based on the argument's
type
    static void print(String value) {
        System.out.println("String: " + value);
    }

    static void print(int value) {
        System.out.println("int: " + value);
    }

    public static void main(String[] args) {
        print("hello"); // resolved at COMPILE time -- calls print(String)
        print(42);      // resolved at COMPILE time -- calls print(int)

        Animal animal = new Dog();
        animal.makeSound(); // resolved at RUNTIME -- calls Dog's makeSound(), based on the
real object
    }
}
```

Which `print(...)` call runs is decided at **compile time**, based on the argument's type — so the compiler bakes that decision straight into the `.class` file, with no ambiguity left at runtime. `animal.makeSound()` is the opposite: the compiler only verifies that the `Animal` type has a `makeSound()` method; which **implementation** actually runs is decided by the JVM at runtime, based on `animal`'s real type (the dynamic dispatch we covered in detail in the Inheritance lesson's "Upcasting" section).

Method Overloading

Defining multiple methods in the same class with the same name but different **parameter lists** (different number, type, or order of parameters) is called **method overloading**. The return type alone isn't enough — two methods that differ only in return type aren't a valid overload, and won't compile:

```
class OverloadingExample {
    static int add(int a, int b) {
        return a + b;
    }

    static double add(double a, double b) {
        return a + b;
    }

    static int add(int a, int b, int c) {
        return a + b + c;
    }

    // static double add(int a, int b) { return a + b; } // compile error: cannot overload
by return type alone

    public static void main(String[] args) {
        System.out.println(add(2, 3));    // 5    -- add(int, int)
        System.out.println(add(2.5, 3.5)); // 6.0  -- add(double, double)
        System.out.println(add(1, 2, 3)); // 6    -- add(int, int, int)
    }
}
```

`add(int, int)`, `add(double, double)`, and `add(int, int, int)` all share the same name, but the compiler decides which one to call by looking at the number and type of arguments given at the call site. This is exactly the compile-time polymorphism we introduced in "What Is Polymorphism?" — there's no runtime cost at all, the decision is made entirely at compile time.

> `@Override` is only ever used for overriding, never for overloading — putting `@Override` on an overload fails to compile, because the compiler expects an actual override there. This is one of the most common misunderstandings about `@Override`.

Overload Resolution Rules

When multiple overload candidates could apply, the compiler picks the "best" one using a specific priority order: first an **exact match**, then **widening** (an implicit conversion from a smaller type to a larger one, like `int` → `long`), then **autoboxing/unboxing** (`int` → `Integer`), and finally **varargs**. Not knowing this order makes it hard to predict which overload actually gets called:

```
class OverloadResolutionExample {
    static void process(int value) {
        System.out.println("process(int): " + value);
    }

    static void process(long value) {
        System.out.println("process(long): " + value);
    }

    static void process(Integer value) {
        System.out.println("process(Integer): " + value);
    }

    static void process(int... values) {
        System.out.println("process(int...): " + values.length + " values");
    }

    public static void main(String[] args) {
        short s = 5;
        process(s); // no process(short) -- WIDENS to int -- process(int)

        process(5L); // exact match -- process(long)

        Integer boxed = 5;
        process(boxed); // exact match -- process(Integer)

        process(1, 2, 3); // no 3-arg overload -- falls back to VARARGS -- process(int...)
    }
}
```

The call `process(s)` has no `process` overload for `short` — the compiler **widens** it to `int` and calls `process(int)`. `process(5L)` and `process(boxed)` **exactly match** the `long` and `Integer` parameters respectively, requiring no conversion at all. `process(1, 2, 3)` has no fixed-parameter overload that takes three `int`s, so it falls back to **varargs** (`process(int...)`) as a last resort.

> Varargs (`int...`) is always considered a **last resort** — if any other overload matches via an exact match or via widening/autoboxing, the compiler won't even try varargs. That's why overloading a method with both fixed parameters and varargs can lead to surprising call resolution.

Covariant Return Types

We already covered the mechanics of method overriding (signature matching, `@Override`, dynamic dispatch) in the Inheritance lesson's "Method Overriding" section — we won't repeat that here. But one rule never came up there: **covariant return types**. An overriding method is allowed to return a **subtype** of the type the superclass method returns — the return type doesn't have to match exactly:

```
class Animal {
    Animal reproduce() {
        return new Animal();
    }
}

class Dog extends Animal {
    @Override
    Dog reproduce() { // covariant return type -- Dog is a subtype of Animal, this is legal
        return new Dog();
    }
}

class CovariantReturnTypeExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Dog puppy = dog.reproduce(); // no cast needed -- reproduce() already returns Dog
here
        System.out.println(puppy.getClass().getSimpleName()); // Dog
    }
}
```

`Animal.reproduce()` returns an `Animal`, but `Dog`'s override of `reproduce()` returns the more specific `Dog` type — this doesn't violate the overriding rules, because every `Dog` already is an `Animal` (the is-a relationship from the Inheritance lesson's "What Is Inheritance?" section). That means code calling it on a `Dog` can use the result directly as a `Dog`, with no cast needed.

Polymorphism vs. Inheritance

These two concepts get used interchangeably a lot, but they're not the same thing: **inheritance is a structural relationship** (one class being derived from another), while **polymorphism is a runtime behavior** (the same call being able to run different implementations). Inheritance **makes polymorphism possible**, but it doesn't **guarantee** it:

```
class Animal {
    void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Cat extends Animal {
    // inherits makeSound() as-is -- no override, so no real polymorphism here
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!"); // overridden -- this IS polymorphism
    }
}

class PolymorphismVsInheritanceExample {
    public static void main(String[] args) {
        Animal cat = new Cat();
        cat.makeSound(); // Some generic animal sound -- inheritance without polymorphism

        Animal dog = new Dog();
        dog.makeSound(); // Woof! -- inheritance WITH polymorphism
    }
}
```

`Cat` inherits from `Animal` (inheritance is present) but never overrides `makeSound()` — so `cat.makeSound()` always runs `Animal`'s behavior; there's no real polymorphism happening here. `Dog`, on the other hand, both inherits and overrides — the actual polymorphism shows up right where the overridden method gets called. The reverse is also possible: as we saw in the Interface lesson, you can get polymorphism with no inheritance hierarchy at all (two unrelated classes implementing the same interface) — polymorphism doesn't **depend on** inheritance, it's just one of the most common places you'll find it.

Polymorphism with Interfaces and Abstract Classes

As we saw in the Interface and Abstract Class lessons (recall the Abstract Class lesson's "Abstract Class vs. Interface" section), polymorphism can be achieved with either — the difference is whether shared implementation is involved. The JDK's `Comparable` interface is a good example of how even completely unrelated types can behave polymorphically through the same interface:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Money implements Comparable<Money> {
    int cents;

    Money(int cents) {
        this.cents = cents;
    }

    @Override
    public int compareTo(Money other) {
        return Integer.compare(this.cents, other.cents);
    }

    @Override
    public String toString() {
        return cents + " cents";
    }
}

class ComparableExample {
    public static void main(String[] args) {
        List<Money> wallet = new ArrayList<>();
        wallet.add(new Money(500));
        wallet.add(new Money(100));
        wallet.add(new Money(250));

        Collections.sort(wallet); // sort() only relies on compareTo() -- doesn't know
Money exists
        System.out.println(wallet); // [100 cents, 250 cents, 500 cents]
    }
}
```

If some completely unrelated type with no common superclass also implemented `Comparable`, `Collections.sort(...)` could sort it the exact same way — because `sort` never knows the actual type of its elements, it just relies on the `compareTo()` contract. This is a concrete application of "program to an interface" (recall the Interface lesson's "Why Does It Exist?" section).

Polymorphism with Composition

The Inheritance lesson's "Inheritance vs. Composition" section showed composition using a concrete class (`Engine`). When composition is paired with an **interface type** instead, a much more powerful pattern emerges: the **Strategy pattern** — a class **holds** a reference to an interface whose behavior can change, rather than implementing that behavior itself:

```
interface TextFormatter {
    String format(String text);
}

class UpperCaseFormatter implements TextFormatter {
    @Override
    public String format(String text) {
        return text.toUpperCase();
    }
}

class ReverseFormatter implements TextFormatter {
    @Override
    public String format(String text) {
        return new StringBuilder(text).reverse().toString();
    }
}

class Document {
    private TextFormatter formatter; // composition -- holds a strategy, doesn't implement
one

    Document(TextFormatter formatter) {
        this.formatter = formatter;
    }

    void setFormatter(TextFormatter formatter) {
        this.formatter = formatter; // behavior can change at runtime
    }

    String render(String text) {
        return formatter.format(text); // delegates -- doesn't know WHICH formatter it's
using
    }
}

class TextFormatterStrategyExample {
    public static void main(String[] args) {
        Document doc = new Document(new UpperCaseFormatter());
        System.out.println(doc.render("hello")); // HELLO

        doc.setFormatter(new ReverseFormatter()); // swap strategy at runtime
        System.out.println(doc.render("hello")); // olleh
    }
}
```

`Document` has no idea which `TextFormatter` implementation it's using — it only relies on the interface's `format(String)` contract. `setFormatter(...)` lets us **swap the behavior at runtime**, something inheritance could never do (an object's class can't change at runtime). This shows why composition becomes so flexible once it's paired with polymorphism.

instanceof: When to Use It, When It's a Code Smell

We already covered `instanceof`'s mechanics (pattern matching, the `ClassCastException` risk) in the Inheritance lesson's "Downcasting and instanceof" section — the real question here is **when you should use it**. If you see a chain of `if (obj instanceof TypeA) {...} else if (obj instanceof TypeB) {...}`, that's usually a sign polymorphism **isn't being used** — because in a well-designed system, calling code never asks "what type is this?", it just calls the polymorphic method directly:

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    String describe() {
        return name + " makes some sound";
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    String describe() {
        return name + " barks";
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    @Override
    String describe() {
        return name + " meows";
    }
}

class InstanceofDesignExample {
    // Brittle: every new Animal subtype means editing this method again
    static String describeWithInstanceof(Animal animal) {
        if (animal instanceof Dog dog) {
            return dog.name + " barks";
        } else if (animal instanceof Cat cat) {
            return cat.name + " meows";
        } else {
            return animal.name + " makes some sound";
        }
    }
```

```
    // Polymorphic: adding a new subtype never requires touching this method
    static String describeWithPolymorphism(Animal animal) {
        return animal.describe();
    }

    public static void main(String[] args) {
        Animal[] animals = { new Dog("Rex"), new Cat("Whiskers") };

        for (Animal a : animals) {
            System.out.println(describeWithInstanceof(a));
            System.out.println(describeWithPolymorphism(a));
        }
    }
}
```

`describeWithInstanceof(...)` is an `if/else` chain that has to grow every time a new animal type is added — adding a new `Bird` means **finding and updating** this method. `describeWithPolymorphism(...)` is a single line: every `Animal` already knows how to produce its own `describe()`, and calling code does no type checking at all. `instanceof` does have **legitimate** uses — filtering out a specific type from a collection of mixed types, or enforcing type safety at an API boundary (as inside `equals(Object)`) — but if it's being used to express a difference in behavior, almost always polymorphism is the better fix.

> If adding a new subtype means updating **every branch** of an `instanceof` chain, that's a strong sign the design needs polymorphism — we'll revisit this pattern in "Common Mistakes."

Polymorphism in Collections

It's no accident that you almost always declare Java collections by their interface type (`List<String> list = new ArrayList<>();`) — this is the most common application you'll run into of the "program to an interface" principle from "Polymorphism with Interfaces and Abstract Classes":

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class CollectionPolymorphismExample {
    // Declared with the INTERFACE type -- works with any List implementation
    static void printAll(List<String> list) {
        for (String s : list) {
            System.out.println(s);
        }
    }

    public static void main(String[] args) {
        List<String> arrayBacked = new ArrayList<>();
        arrayBacked.add("a");
        arrayBacked.add("b");

        List<String> linkedBacked = new LinkedList<>();
        linkedBacked.add("c");
        linkedBacked.add("d");

        printAll(arrayBacked);  // same method, different List implementation
        printAll(linkedBacked); // same method, different List implementation
    }
}
```

The `printAll(List<String> list)` method has no idea — and doesn't need to know — whether it received an `ArrayList` or a `LinkedList`; it only relies on the `List` contract. That means swapping the implementation (say, going from `ArrayList` to `LinkedList`) doesn't affect **a single line** of the calling code — a reflection of the flexibility we saw in "Polymorphism with Composition," applied to collections.

Real-World Examples

Polymorphism is the backbone of nearly every core JDK API. The `InputStream` / `OutputStream` hierarchy in `java.io` is a classic example: code that reads from a file, a network socket, or an in-memory byte array all works **the same way**, through the same `InputStream` interface. We can build the same pattern in a small example:

```
interface DataSource {
    String readData();
}

class FileDataSource implements DataSource {
    @Override
    public String readData() {
        return "data read from a file";
    }
}

class MemoryDataSource implements DataSource {
    private String data;

    MemoryDataSource(String data) {
        this.data = data;
    }

    @Override
    public String readData() {
        return data;
    }
}

class RealWorldPolymorphismExample {
    // Mirrors how code that reads from an InputStream never cares about the real source
    static void readAll(DataSource source) {
        System.out.println(source.readData());
    }

    public static void main(String[] args) {
        readAll(new FileDataSource());             // data read from a file
        readAll(new MemoryDataSource("in-memory")); // in-memory
    }
}
```

`readAll(DataSource source)` has no idea whether the data is coming from a file or from memory — just like code calling `read()` on a real `InputStream` doesn't need to know where the data is actually coming from. The same idea plays a central role in the Spring framework: a `@Service` class takes the dependency it needs as an **interface** type rather than a concrete class (constructor injection); Spring decides at runtime which implementation (`StripePaymentService`, `PaypalPaymentService`, and so on) to inject. That's nothing more than the Strategy pattern from "Polymorphism with Composition," automated by a framework.

Best Practices

- If you see an `if/else` / `switch` chain that keeps growing with every new type, consider replacing it with an interface plus polymorphism (see "instanceof: When to Use It, When It's a Code Smell").

- Declare method parameters and collection variables by their **interface type** whenever possible (`List` , not `ArrayList`) — it makes swapping implementations much easier (see "Polymorphism in Collections").
- When designing behavior that might change at runtime, reach for composition + an interface (the Strategy pattern) rather than locking it in with inheritance (see "Polymorphism with Composition").
- When designing an overload set, test what each argument resolves to from the compiler's point of view — avoid writing ambiguous overloads (see "Overload Resolution Rules").
- Don't hesitate to narrow an overriding method's return type with a covariant return type when it genuinely returns something more specific (see "Covariant Return Types").

Common Mistakes

1. Manually updating a chain of `instanceof` checks every time a new type is added. This is exactly the problem polymorphism is meant to solve (see "instanceof: When to Use It, When It's a Code Smell").

2. Trying to overload a method by changing only its return type. Java can't distinguish two methods by return type alone — the parameter list has to differ too (see "Method Overloading").

3. Designing ambiguous overloads without knowing the overload resolution order. Not knowing the order in which widening, autoboxing, and varargs are tried leads to guessing wrong about which method actually gets called (see "Overload Resolution Rules").

4. Assuming inheritance automatically gives you polymorphism. If a subclass never overrides any method, there's no real polymorphism there — just inheritance (see "Polymorphism vs. Inheritance").

5. Declaring collection variables by their concrete implementation type (`ArrayList<String> list = new ArrayList<>();`). This throws away the flexibility to swap implementations later (see "Polymorphism in Collections").

Summary, Cheat Sheet, and Glossary

Polymorphism is Java's name for "same interface, different implementations," and it comes in two forms — resolved at compile time (overloading) and at runtime (overriding). Key takeaways:

- **Compile-time polymorphism (method overloading):** The compiler decides which method runs by looking at the number and type of arguments; return type alone can't distinguish an overload
- Overload resolution order: **exact match → widening → autoboxing/unboxing → varargs**

- **Runtime polymorphism (method overriding):** The JVM decides which implementation runs at runtime, based on the object's actual type (dynamic dispatch)
- An overriding method can return a subtype of what the superclass method returns (**covariant return type**)
- Inheritance is a structural relationship, polymorphism is a runtime behavior — if a subclass never overrides anything, there's inheritance but no polymorphism
- Composition paired with an interface gives you the **Strategy pattern** — behavior becomes swappable at runtime
- A growing `instanceof` / `switch` chain is usually a sign of a missing polymorphic design
- Declaring collections by their interface type (`List` , `Set` , `Map`) keeps you free to swap implementations
- The JDK (`InputStream` / `OutputStream` , `Comparable`) and Spring (interface-based dependency injection) both lean heavily on polymorphism

Quick reference:

```
// Compile-time polymorphism -- overloading
static void print(String value) { }
static void print(int value) { }
print("a"); // resolved at compile time by argument type
print(1);

// Runtime polymorphism -- overriding
class Animal {
    void makeSound() { }
}
class Dog extends Animal {
    @Override
    void makeSound() { }         // resolved at runtime by the object's real type
}
Animal a = new Dog();
a.makeSound();

// Covariant return type
class Animal2 {
    Animal2 reproduce() { return new Animal2(); }
}
class Dog2 extends Animal2 {
    @Override
    Dog2 reproduce() { return new Dog2(); } // narrower return type -- legal
}

// Strategy pattern -- composition + polymorphism
interface Formatter { String format(String s); }
class Document {
    private Formatter formatter; // held, not extended
    Document(Formatter formatter) { this.formatter = formatter; }
    String render(String s) { return formatter.format(s); }
}

// instanceof chain vs polymorphism
// Bad:
// if (obj instanceof Dog d) { ... } else if (obj instanceof Cat c) { ... }
// Good:
// obj.describe(); // let the object decide
```

Glossary

Polymorphism — The ability of the same interface (method call) to produce different behavior depending on the object's actual type.

Compile-time polymorphism — The kind of polymorphism where the compiler decides which method runs by looking at the arguments' types; this is method overloading.

Runtime polymorphism — The kind of polymorphism where the JVM decides which implementation runs at runtime, based on the object's actual type; this is method overriding.

Method overloading — Defining multiple methods in the same class with the same name but different parameter lists.

Overload resolution order — The priority the compiler follows when choosing among overload candidates: exact match, widening, autoboxing/unboxing, varargs.

Covariant return type — The ability of an overriding method to return a subtype of what the superclass method returns.

Strategy pattern — A design pattern where a class holds a reference to an interface whose behavior can be swapped at runtime, instead of extending a class; composition combined with polymorphism.

Dynamic dispatch — Deciding which implementation of a method call runs based on the object's actual runtime type, rather than at compile time.

Appendix: Mini Project — From if/else Chains to Polymorphism: Discount Calculation

In this mini project we carry out end-to-end the refactor described in "instanceof: When to Use It, When It's a Code Smell": we turn a fragile `if/else` chain that calculates a discount based on customer type into a polymorphic design built around a `DiscountStrategy` interface:

```
class LegacyDiscountCalculator {
    // The brittle version: every new customer type means editing this method
    static double calculateDiscountWithIfElse(String customerType, double amount) {
        if (customerType.equals("VIP")) {
            return amount * 0.20;
        } else if (customerType.equals("REGULAR")) {
            return amount * 0.05;
        } else {
            return 0.0;
        }
    }
}

interface DiscountStrategy {
    double apply(double amount);
}

class PercentageDiscount implements DiscountStrategy {
    private double percentage;

    PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double apply(double amount) {
        return amount * percentage;
    }
}

class FlatDiscount implements DiscountStrategy {
    private double flatAmount;

    FlatDiscount(double flatAmount) {
        this.flatAmount = flatAmount;
    }

    @Override
    public double apply(double amount) {
        return Math.min(flatAmount, amount);
    }
}

class NoDiscount implements DiscountStrategy {
    @Override
    public double apply(double amount) {
        return 0.0;
    }
}
```

```
class DiscountStrategyDemo {
    public static void main(String[] args) {
        System.out.println(LegacyDiscountCalculator.calculateDiscountWithIfElse("VIP",
200)); // 40.0

        DiscountStrategy[] strategies = {
            new PercentageDiscount(0.20), // VIP-style
            new FlatDiscount(10),         // coupon-style
            new NoDiscount()              // no discount
        };

        for (DiscountStrategy strategy : strategies) {
            System.out.println(strategy.apply(200)); // 40.0 / 10.0 / 0.0
        }
    }
}
```

`calculateDiscountWithIfElse(...)` is a chain that has to grow with every new customer type; `PercentageDiscount`, `FlatDiscount`, and `NoDiscount` each implement `DiscountStrategy`'s `apply(double)` contract with their own formula. The loop in `DiscountStrategyDemo` calls `apply(...)` without ever knowing which strategy it's holding — adding a new kind of discount now just means writing a new class, with no existing code to touch.

> This principle is known as the **Open/Closed Principle**: code should be **open to extension** but **closed to modification**. Polymorphism is the most common way to put that principle into practice.

Appendix: Mini Project — Composition + Strategy: A Notification-Sending System

Our last mini project carries the idea from "Polymorphism with Composition" into a more realistic scenario: a `NotificationService` has no idea which channel (email, SMS, push) it's sending through — it gets that through **composition**, via a `NotificationSender` interface:

```
interface NotificationSender {
    void send(String message);
}

class EmailSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Email sent: " + message);
    }
}

class SmsSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("SMS sent: " + message);
    }
}

class PushSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Push notification sent: " + message);
    }
}

class NotificationService {
    private NotificationSender sender; // composition -- holds a channel, doesn't become one

    NotificationService(NotificationSender sender) {
        this.sender = sender;
    }

    void setSender(NotificationSender sender) {
        this.sender = sender; // channel can be swapped at runtime
    }

    void notifyUser(String message) {
        sender.send(message); // delegates -- doesn't know which channel it's using
    }
}
```

```
class NotificationSenderDemo {
    public static void main(String[] args) {
        NotificationService service = new NotificationService(new EmailSender());
        service.notifyUser("Your order has shipped"); // Email sent: ...

        service.setSender(new SmsSender()); // swap channel at runtime
        service.notifyUser("Your order has shipped"); // SMS sent: ...

        service.setSender(new PushSender());
        service.notifyUser("Your order has shipped"); // Push notification sent: ...
    }
}
```

`NotificationService` **takes** a `NotificationSender` in its constructor (composition) rather than extending one. `EmailSender`, `SmsSender`, and `PushSender` each implement the same `send(String)` contract their own way. As `NotificationSenderDemo` shows, the same `NotificationService` object can switch to a different channel **at runtime** via `setSender(...)` — a flexibility inheritance could never give you. That's the core advantage we highlighted in "Polymorphism with Composition."

> If we'd designed `NotificationService` to extend `EmailSender` (even though the "is-a" relationship never made sense there), every new channel would mean either writing a new `NotificationService` subclass or bloating a single class to support every channel — exactly the trap the Inheritance lesson's "Inheritance vs. Composition" section warned about.