

The Problem with new

- new requires a concrete class
- every client that creates objects must know
 - which concrete class to pick
 - how to construct it
- a change to either means editing every client
- we can't eliminate needing a concrete class for new but we can move the creation decisions to one place

Factories

Intent –

- hide which concrete class is instantiated (and how) from the code that uses the object

Variants –

- Simple factory (a common idiom, not an official pattern)
 - use when many places need the same kind of object
 - creation step is encapsulated in a method which picks the product based on a parameter
- Factory Method
 - use when a base class or framework owns the overall algorithm but the product for one step varies
 - creation step is encapsulated in a method – subclass creates the specific product
 - supports OCP – add new products by writing new creator subclasses
- Abstract Factory
 - use when several kinds of objects must be used together
 - factory contains one creation method per product kind
 - client is given the factory – swap the factory to swap the whole family at once

Simple Factory

- a static method returns an interface type, hiding which concrete class it picks
- not an official pattern, but the most common form in practice

```
public class PaymentStrategies {
    public static PaymentStrategy forType ( String type ) {
        if ( type.equals("card") ) {
            return new CreditCardPayment();
        } else if ( type.equals("paypal") ) {
            return new PayPalPayment();
        } else {
            throw new IllegalArgumentException("Unknown: " + type);
        }
    }
}

// client
PaymentStrategy p = PaymentStrategies.forType(userChoice);
cart.checkout(p);
```

- violates OCP, but at least the if is limited to one place

Registry-Based Simple Factory

- OCP fix – a supplier registry

```
public class PaymentStrategies {
    private static final Map<String, Supplier<PaymentStrategy>> REGISTRY =
        new HashMap<>();

    static {
        REGISTRY.put("card", CreditCardPayment::new);
        REGISTRY.put("paypal", PayPalPayment::new);
    }

    public static PaymentStrategy forType ( String type ) {
        Supplier<PaymentStrategy> maker = REGISTRY.get(type);
        if ( maker == null ) {
            throw new IllegalArgumentException("Unknown: " + type);
        }
        return maker.get();
    }

    public static void register(String type, Supplier<PaymentStrategy> maker) {
        REGISTRY.put(type, maker);
    }
}

// client
PaymentStrategy p = PaymentStrategies.forType(userChoice);

// extension: no edit to PaymentStrategies' logic
PaymentStrategies.register("crypto", CryptoPayment::new);
```

Supplier<T>

- Supplier<T> is a Java functional interface
 - “something that can produce a T when you ask”

```
public interface Supplier<T> {
    T get();
}
```

- writing a Supplier

```
// 1. Lambda
Supplier<PaymentStrategy> a = () -> new CreditCardPayment();

// 2. Constructor reference (shorthand for the lambda above)
Supplier<PaymentStrategy> b = CreditCardPayment::new;

// 3. Anonymous class
Supplier<PaymentStrategy> c = new Supplier<PaymentStrategy>() {
    public PaymentStrategy get() { return new CreditCardPayment(); }
};

// All three are used the same way
PaymentStrategy p = a.get(); // each call runs the constructor
PaymentStrategy q = a.get(); // a different object
```

151

Registry-Based Simple Factory

- what it gets you
 - OCF for the factory – a new type means a call to register rather than having to change the factory itself
 - no growing if statement
 - other code (or a test) can register its own implementation
- caveats and drawbacks
 - the registry is mutable global state – anyone can call register
 - can impose stricter control by only populating at startup, but then others can't register their own implementations
 - unknown keys fail at runtime
 - also true of the big if version, but the big if at least contained all of the cases in one place
 - more layers of indirection
 - over-designed if there are only a few different types

CPSC 329: Software Development • Fall 2026

151

Factory and Strategy Together

- Strategy says what gets plugged in, Factory says who builds it
 - the factory chooses the concrete strategies
 - the client asks for a configured object and never sees the concrete classes

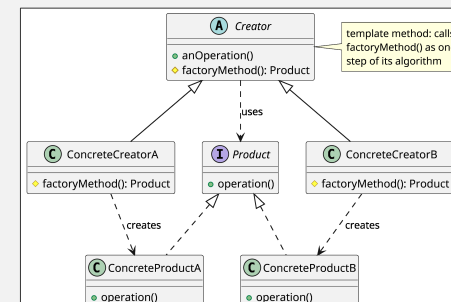
```
public static LoanHandler createHandler ( LoanType type ) {
    if ( type == CAR ) {
        return new LoanHandler(new CarLoanValidator(),new PrintOutcome());
    } else if ( type == MORTGAGE ) {
        return new LoanHandler(new MortgageValidator(),new PrintOutcome());
    } else {
        throw new IllegalArgumentException();
    }
}
```

CPSC 329: Software Development • Fall 2026

153

Factory Method

- base class has a template method which encapsulates creation logic
- subclasses determine the product



CPSC 329: Software Development • Fall 2026

154

Factory Method Example

```
public interface Transport {
    void deliver(String cargo);
}

public class Truck implements Transport {
    public void deliver(String cargo) {
        System.out.println("Truck delivers " + cargo + " by road");
    }
}

public class Ship implements Transport {
    public void deliver(String cargo) {
        System.out.println("Ship delivers " + cargo + " by sea");
    }
}
```

Factory Method Example

```
public abstract class Logistics {
    protected abstract Transport createTransport(); // factory method

    public void planDelivery(String cargo) { // template method
        Transport t = createTransport();
        System.out.println("Loading " + cargo);
        t.deliver(cargo);
        System.out.println("Delivery complete");
    }
}

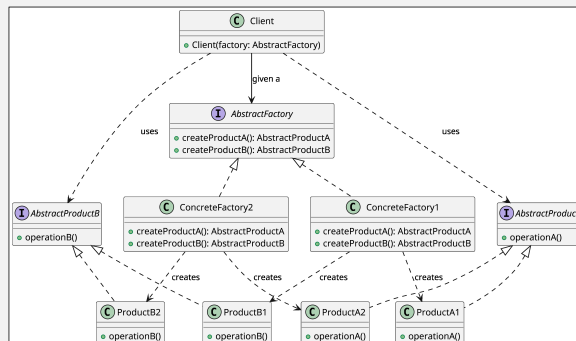
public class RoadLogistics extends Logistics {
    public RoadLogistics() { }
    protected Transport createTransport() { return new Truck(); }
}

public class SeaLogistics extends Logistics {
    public SeaLogistics() { }
    protected Transport createTransport() { return new Ship(); }
}

// client
Logistics l = new SeaLogistics();
l.planDelivery("bananas");
```

Abstract Factory

- commonly built of factory methods, one per product kind
 - each product family has a full set covering that family's version of each product



Abstract Factory Example

- UI components

```
public interface Button { void paint(); }
public interface Checkbox { void paint(); }

public class WinButton implements Button {
    public WinButton() { ... }
    public void paint() { ... }
}

public class WinCheckbox implements Checkbox {
    public WinCheckbox() { ... }
    public void paint() { ... }
}

public class MacButton implements Button {
    public MacButton() { ... }
    public void paint() { ... }
}

public class MacCheckbox implements Checkbox {
    public MacCheckbox() { ... }
    public void paint() { ... }
}
```

```
public interface GUIFactory {           // the abstract factory
    Button  createButton();
    Checkbox createCheckbox();
}

public class WinFactory implements GUIFactory {
    public WinFactory() { }
    public Button  createButton() { return new WinButton(); }
    public Checkbox createCheckbox() { return new WinCheckbox(); }
}

public class MacFactory implements GUIFactory {
    public MacFactory() { }
    public Button  createButton() { return new MacButton(); }
    public Checkbox createCheckbox() { return new MacCheckbox(); }
}

public class SettingsDialog {
    private final Button ok;
    private final Checkbox remember;
    public SettingsDialog(GUIFactory f) { // client sees only interfaces
        ok = f.createButton();
        remember = f.createCheckbox();
    }
}

// chosen once, at startup
GUIFactory f = isMac() ? new MacFactory() : new WinFactory();
new SettingsDialog(f);
```

Factories Summary

	simple factory	Factory Method	Abstract Factory
pattern or idiom?	idiom	pattern	pattern
what it creates	one product type, any of several variants	one product, via an overridable method	a family of related products that must match
who decides which variant	a conditional inside the factory, driven by a parameter	the subclass you instantiated	the factory object you were given
mechanism	a static method	inheritance – a hook in a base class	composition – a factory object passed to the client
typical shape	ShapeFactory.create("circle")	abstract createTransport() in Logistics	interface GUIFactory { createButton(); createCheckbox(); }



Decorator

Intent –

- add behavior to an individual object

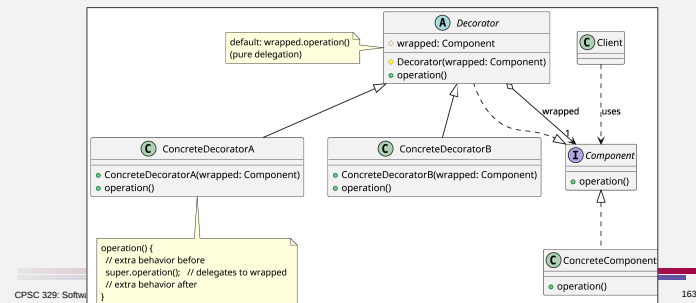
The problems –

- avoiding a subclass explosion when there are many combinations
- inheritance fixes the combination at compile time

Decorator

Structure –

- a decorator implements the same interface as the thing it wrap and holds a reference to one wrapped object
- behavior is added before or after delegating to the wrapped object
- decorators stack, so clients can't tell a decorated object from a plain one



Decorator Example

```
public interface Beverage {
    String description();
    double cost();
}

public class Espresso implements Beverage {
    public Espresso() {}
    public String description() { return "Espresso"; }
    public double cost() { return 1.99; }
}

public class HouseBlend implements Beverage {
    public HouseBlend() {}
    public String description() { return "House Blend"; }
    public double cost() { return 0.89; }
}
```

Decorator Example

```
public abstract class CondimentDecorator implements Beverage {
    protected final Beverage wrapped;
    protected CondimentDecorator(Beverage wrapped) { this.wrapped = wrapped; }
}

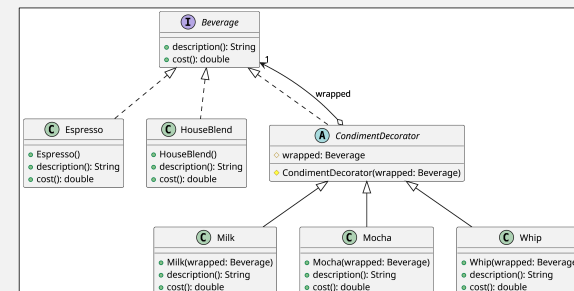
public class Milk extends CondimentDecorator {
    public Milk(Beverage wrapped) { super(wrapped); }
    public String description() { return wrapped.description() + ", Milk"; }
    public double cost() { return wrapped.cost() + 0.10; }
}

public class Mocha extends CondimentDecorator {
    public Mocha(Beverage wrapped) { super(wrapped); }
    public String description() { return wrapped.description() + ", Mocha"; }
    public double cost() { return wrapped.cost() + 0.20; }
}

public class Whip extends CondimentDecorator {
    public Whip(Beverage wrapped) { super(wrapped); }
    public String description() { return wrapped.description() + ", Whip"; }
    public double cost() { return wrapped.cost() + 0.15; }
}
```

Decorator Example

```
// client: a double mocha with whip
Beverage b = new Whip(new Mocha(new Mocha(new Espresso())));
System.out.println(b.description() + " $" + b.cost());
// Espresso, Mocha, Mocha, Whip $2.54
```



Decorator vs Strategy vs subclassing

	structure	intent
subclassing	behavior fixed at compile time by the class chosen	one variation, set once
Strategy	context holds an object of a different interface and calls it	choose or replace one piece of behavior
Decorator	wrapper implements the same interface as the object it holds, and delegates to it	add behavior around existing behavior stackable