```java
/**
 * State pattern (cat-themed)
 *
 * Case-study notes:
 * - Intent: Allow an object (a cat) to alter its behavior when its internal
 * state
 * changes (Idle, Eating, Sleeping) without large conditional statements.
 * - When to use: modeling objects with distinct modes of behavior that change
 * at runtime.
 * - Participants in this example:
 * - Context: `AnimalContext` (holds current state)
 * - State interface: `AnimalStateBehavior`
 * - ConcreteStates: `IdleState`, `EatingState`, `SleepingState`
 *
 * The cat example demonstrates how calls like `performEat` or `performSound`
 * produce different outputs depending on the current state.
 *
 *
 * Simple cat State example: Idle -> Eating -> Sleeping
 */
public class AnimalState {
    public static void main(String[] args) {
        AnimalContext kitty = new AnimalContext(new IdleState());

        kitty.performSound(); // Meow
        kitty.performEat(); // starts eating (transitions to Eating)
        kitty.performSound(); // Purr
        kitty.performSleep(); // finishes and sleeps
        kitty.performSound(); // Soft snore
    }
}

class AnimalContext {
    private AnimalStateBehavior state;

    public AnimalContext(AnimalStateBehavior state) {
        this.state = state;
    }

    public void setState(AnimalStateBehavior s) {
        this.state = s;
    }

    public void performEat() {
        state.eat(this);
    }

    public void performSleep() {
        state.sleep(this);
    }

    public void performSound() {
        state.sound(this);
    }
}

interface AnimalStateBehavior {
    void eat(AnimalContext ctx);

    void sleep(AnimalContext ctx);

    void sound(AnimalContext ctx);
}

class IdleState implements AnimalStateBehavior {
    public void eat(AnimalContext ctx) {
        System.out.println("Kitty starts nibbling (idle->eating)");
        ctx.setState(new EatingState());
    }

    public void sleep(AnimalContext ctx) {
        System.out.println("Kitty curls up (idle->sleeping)");
        ctx.setState(new SleepingState());
    }

    public void sound(AnimalContext ctx) {
        System.out.println("Meow");
    }
}

class EatingState implements AnimalStateBehavior {
    public void eat(AnimalContext ctx) {
        System.out.println("Kitty is eating: crunch");
    }

    public void sleep(AnimalContext ctx) {
        System.out.println("Kitty finishes and dozes off");
        ctx.setState(new SleepingState());
    }

    public void sound(AnimalContext ctx) {
        System.out.println("Purr");
    }
}

class SleepingState implements AnimalStateBehavior {
    public void eat(AnimalContext ctx) {
        System.out.println("Kitty is asleep; won't eat now");
    }

    public void sleep(AnimalContext ctx) {
        System.out.println("Kitty snores softly");
    }

    public void sound(AnimalContext ctx) {
        System.out.println("Soft snore");
    }
}
```

```java
/**
 * Strategy pattern (cat-themed)
 *
 * Case-study notes:
 * - Intent: Define a family of algorithms (feeding behaviors) and make them
 * interchangeable at runtime.
 * - When to use: when the same task (feed) can be done in different ways
 * (kibble, fish, treats) and you want to swap behaviors without changing the
 * client.
 * - Participants in this example:
 * - Strategy: `FeedingStrategy` (interface)
 * - ConcreteStrategies: `KibbleFeeding`, `FishFeeding`, `TreatFeeding`
 * - Context: `Animal` (uses a `FeedingStrategy` to feed itself)
 *
 * The cat example shows named cats using different feeding strategies and
 * swapping them at runtime.
 */
public class AnimalStrategy {
    public static void main(String[] args) {
        Animal whiskers = new Animal("Whiskers", new KibbleFeeding());
        Animal mittens = new Animal("Mittens", new FishFeeding());
        Animal luna = new Animal("Luna", new TreatFeeding());

        whiskers.feed("dry kibble");
        mittens.feed("salmon");
        luna.feed("catnip treat");

        // Swap strategy at runtime (e.g., switching diet)
        System.out.println("-- Switching Whiskers to Fish diet --");
        whiskers.setFeedingStrategy(new FishFeeding());
        whiskers.feed("tuna");
    }
}

interface FeedingStrategy {
    void eat(String food);
}

class KibbleFeeding implements FeedingStrategy {
    public void eat(String food) {
        System.out.println("Eats kibble: " + food);
    }
}

class FishFeeding implements FeedingStrategy {
    public void eat(String food) {
        System.out.println("Eats fish: " + food);
    }
}

class TreatFeeding implements FeedingStrategy {
    public void eat(String food) {
        System.out.println("Enjoys treat: " + food);
    }
}

class Animal {
    private String name;
    private FeedingStrategy feedingStrategy;

    public Animal(String name, FeedingStrategy fs) {
        this.name = name;
        this.feedingStrategy = fs;
    }

    public void setFeedingStrategy(FeedingStrategy fs) {
        this.feedingStrategy = fs;
    }

    public void feed(String food) {
        System.out.print(name + " -> ");
        feedingStrategy.eat(food);
    }
}

```

```java
/**
 * Adapter pattern (cat-themed)
 *
 * Case-study notes:
 * - Intent: Convert the interface of a class (here `OldCat`) into another
 * interface
 * (`Pet`) clients expect without modifying the original class.
 * - When to use: integrating legacy or third-party code with your own
 * interfaces.
 * - Participants in this example:
 * - Target: `Pet` (the interface we want to use)
 * - Adaptee: `OldCat` (legacy class we can't change)
 * - Adapter: `OldPetAdapter` (wraps `OldCat` and implements `Pet`)
 *
 * This demo shows adapting an old cat API to a simple `Pet` interface.
 */
public class AnimalAdapter {
    public static void main(String[] args) {
        OldCat old = new OldCat();
        Pet pet = new OldPetAdapter(old);

        // Client code uses the `Pet` interface and is unaware of `OldCat` internals
        pet.sound();
        pet.eat("kibble");
    }
}

// Desired interface
interface Pet {
    void sound();
    void eat(String food);
}

// Legacy class we cannot change - represents an old cat API
class OldCat {
    void meow() {
        System.out.println("OldCat: meow-meow (old style)");
    }

    void nibble(String food) {
        System.out.println("OldCat nibbles: " + food);
    }
}

// Adapter translates Pet calls to OldCat methods
class OldPetAdapter implements Pet {
    private OldCat oldCat;

    public OldPetAdapter(OldCat oldCat) {
        this.oldCat = oldCat;
    }

    public void sound() {
        oldCat.meow();
    }

    public void eat(String food) {
        oldCat.nibble(food);
    }
}
```

```java
/**
 * Template Method pattern (cat-themed)
 *
 * Case-study notes:
 * - Intent: Define the skeleton of an algorithm (daily routine) in a base class
 * while allowing subclasses to override specific steps.
 * - When to use: when you have a fixed sequence of steps but some steps vary
 * between implementations (e.g., HouseCat vs OutsideCat routines).
 * - Participants in this example:
 * - AbstractClass: `AnimalTemplate` (defines `dailyRoutine` template method)
 * - ConcreteClass: `HouseCatRoutine`, `OutsideCatRoutine` (provide step
 * implementations)
 *
 * The cat routines illustrate two different concrete behaviors while preserving
 * the same routine order.
 */
public abstract class AnimalTemplate {
    public final void dailyRoutine() {
        wakeUp();
        makeSound();
        eat();
        play();
        sleep();
    }

    protected abstract void wakeUp();

    protected abstract void makeSound();

    protected abstract void eat();

    protected abstract void play();

    protected void sleep() {
        System.out.println("Zzz...");
    }
}

// Two cat-themed routines to show variation
class HouseCatRoutine extends AnimalTemplate {
    protected void wakeUp() {
        System.out.println("House Cat wakes up");
    }

    protected void makeSound() {
        System.out.println("Meow (soft)");
    }

    protected void eat() {
        System.out.println("House Cat licks food from bowl");
    }

    protected void play() {
        System.out.println("House Cat bats a toy mouse");
    }
}

class OutsideCatRoutine extends AnimalTemplate {
    protected void wakeUp() {
        System.out.println("Outside Cat wakes up");
    }

    protected void makeSound() {
        System.out.println("Loud Meow (asking for food)");
    }

    protected void eat() {
        System.out.println("Outside Cat scavenges for food");
    }

    protected void play() {
        System.out.println("Outside Cat chases a rat");
    }
}

class AnimalTemplateDemo {
    public static void main(String[] args) {
        AnimalTemplate house = new HouseCatRoutine();
        AnimalTemplate outside = new OutsideCatRoutine();

        System.out.println("House Cat routine:");
        house.dailyRoutine();

        System.out.println("---");
        System.out.println("Outside Cat routine:");
        outside.dailyRoutine();
    }
}

```