```java
package model;

public class DragonPet extends VirtualPet {
    private final String dragonType;
    private final String breathWeapon;
    private int firepower;
    private boolean canFly;

    // Default constructor
    public DragonPet() {
        super("Dragon", new PetSpecies("Dragon", new String[]{"Egg",
"Hatchling", "Wyrmling", "Adult", "Ancient"}, 5000, "Mountain"));
        this.dragonType = "Fire Dragon";
        this.breathWeapon = "Fire Breath";
        this.firepower = 50;
        this.canFly = false;
    }

    public DragonPet(String petName, String dragonType, String
breathWeapon) {
        super(petName, new PetSpecies("Dragon", new String[]{"Egg",
"Hatchling", "Wyrmling", "Adult", "Ancient"}, 5000, "Mountain"));
        this.dragonType = dragonType;
        this.breathWeapon = breathWeapon;
        this.firepower = calculateInitialFirepower(dragonType);
        this.canFly = false; // Dragons learn to fly as they grow
    }

    // Dragon-specific getters
    public String getDragonType() { return dragonType; }
    public String getBreathWeapon() { return breathWeapon; }
    public int getFirepower() { return firepower; }
    public boolean canFly() { return canFly; }

    // Dragon-specific behaviors
    public void breatheFire() {
        if (getHealth() > 30) {
            System.out.println(getPetName() + " breathes " + breathWeapon
+ " with power " + firepower + "!");
            // Breathing fire consumes energy
```

```
            setHealth(getHealth() - 10);
            // But increases happiness if successful
            setHappiness(Math.min(100, getHappiness() + 15));
        } else {
            System.out.println(getPetName() + " is too weak to breathe
fire!");
        }
    }

    public void learnToFly() {
        if (getAge() >= 100 && !canFly) {
            canFly = true;
            System.out.println(getPetName() + " has learned to fly!");
            setHappiness(Math.min(100, getHappiness() + 30));
        } else if (canFly) {
            System.out.println(getPetName() + " already knows how to
fly!");
        } else {
            System.out.println(getPetName() + " is too young to learn
flying!");
        }
    }

    public void hoardTreasure() {
        System.out.println(getPetName() + " is hoarding treasure!");
        setHappiness(Math.min(100, getHappiness() + 20));
        firepower += 5; // Getting stronger from hoarding
    }

    // Override feeding behavior for dragons
    @Override
    public void feedPet(String foodType) {
        if (foodType.equals("Meat") || foodType.equals("Gold")) {
            super.feedPet(foodType);
            if (foodType.equals("Gold")) {
                firepower += 10; // Gold makes dragons more powerful
                System.out.println(getPetName() + "'s firepower
increased!");
            }
        } else {
```

```
            System.out.println("Dragons prefer meat or gold!");
        }
    }

    // Override play behavior
    @Override
    public void playWithPet(String gameType) {
        if (gameType.equals("Flying") && canFly) {
            setHappiness(Math.min(100, getHappiness() + 25));
            System.out.println(getPetName() + " soars through the
skies!");
        } else if (gameType.equals("Treasure Hunt")) {
            hoardTreasure();
        } else {
            super.playWithPet(gameType);
        }
    }

    private int calculateInitialFirepower(String type) {
        switch (type.toLowerCase()) {
            case "fire dragon": return 60;
            case "ice dragon": return 55;
            case "lightning dragon": return 70;
            default: return 50;
        }
    }

    @Override
    public String toString() {
        return "DragonPet{" + "name='" + getPetName() + "', type='" +
dragonType +
                "', weapon='" + breathWeapon + "', firepower=" + firepower
+
                ", canFly=" + canFly + "}";
    }
}
```

```
package model;
```

```java
import java.util.Arrays;

public final class PetSpecies {
    private final String speciesName;
    private final String[] evolutionStages;
    private final int maxLifespan;
    private final String habitat;

    public PetSpecies(String speciesName, String[] evolutionStages, int
maxLifespan, String habitat) {
        if (speciesName == null || evolutionStages == null ||
evolutionStages.length == 0 || maxLifespan <= 0 || habitat == null) {
            throw new IllegalArgumentException("Invalid species data");
        }
        this.speciesName = speciesName;
        this.evolutionStages = Arrays.copyOf(evolutionStages,
evolutionStages.length);
        this.maxLifespan = maxLifespan;
        this.habitat = habitat;
    }

    public String getSpeciesName() { return speciesName; }
    public String[] getEvolutionStages() { return
Arrays.copyOf(evolutionStages, evolutionStages.length); }
    public int getMaxLifespan() { return maxLifespan; }
    public String getHabitat() { return habitat; }

    @Override
    public String toString() {
        return "PetSpecies{" + "speciesName='" + speciesName + '\'' + ",
habitat='" + habitat + '\'' + '}';
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof PetSpecies)) return false;
        PetSpecies other = (PetSpecies) obj;
        return speciesName.equals(other.speciesName) &&
habitat.equals(other.habitat);
```

```
    }

    @Override
    public int hashCode() {
        return speciesName.hashCode() + habitat.hashCode();
    }
}
```

```
package model;

public class RobotPet extends VirtualPet {
    private boolean needsCharging;
    private int batteryLevel;

    // Default constructor
    public RobotPet() {
        super("Robot Pet", new PetSpecies("Robot", new
String[]{"Assembly", "Boot", "Active", "Advanced"}, 10000, "Laboratory"));
        this.needsCharging = false;
        this.batteryLevel = 100;
    }

    public RobotPet(String petName, PetSpecies species, boolean
needsCharging, int batteryLevel) {
        super(petName, species);
        this.needsCharging = needsCharging;
        setBatteryLevel(batteryLevel);
    }

    // Robot-specific getters and setters
    public boolean isNeedsCharging() { return needsCharging; }
    public void setNeedsCharging(boolean needsCharging) {
this.needsCharging = needsCharging; }

    public int getBatteryLevel() { return batteryLevel; }
    public void setBatteryLevel(int batteryLevel) {
        if (batteryLevel < 0 || batteryLevel > 100) throw new
IllegalArgumentException("Battery out of range");
```

```
        this.batteryLevel = batteryLevel;
        this.needsCharging = batteryLevel < 20; // Auto-set charging need
    }

    // Robot-specific behaviors
    public void charge() {
        if (batteryLevel < 100) {
            batteryLevel = Math.min(100, batteryLevel + 25);
            if (batteryLevel >= 20) {
                needsCharging = false;
            }
            System.out.println(getPetName() + " is charging. Battery: " +
batteryLevel + "%");
        }
    }

    public void performDiagnostics() {
        System.out.println("Running diagnostics on " + getPetName());
        System.out.println("Battery Level: " + batteryLevel + "%");
        System.out.println("Charging needed: " + needsCharging);
        System.out.println("Health: " + getHealth() + "%");
    }

    // Override feeding behavior for robots
    @Override
    public void feedPet(String foodType) {
        if (foodType.equals("Electricity")) {
            charge();
        } else {
            System.out.println("Robots don't eat " + foodType + ". Try
charging instead!");
        }
    }

    // Override play behavior to consume battery
    @Override
    public void playWithPet(String gameType) {
        if (batteryLevel > 10) {
            super.playWithPet(gameType);
            setBatteryLevel(batteryLevel - 5); // Playing consumes battery
```

```
        } else {
            System.out.println(getPetName() + " is too low on battery to
play!");
        }
    }

    @Override
    public String toString() {
        return "RobotPet{" + "name='" + getPetName() + "', batteryLevel="
+ batteryLevel +
                ", needsCharging=" + needsCharging + ", happiness=" +
getHappiness() + "}";
    }
}
```

```
package model;

import java.util.UUID;

public class VirtualPet {
    private final String petId;
    private final PetSpecies species;
    private final long birthTimestamp;

    private String petName;
    private int age;
    private int happiness;
    private int health;

    protected static final String[] DEFAULT_EVOLUTION_STAGES = {"Egg",
"Baby", "Teen", "Adult"};
    static final int MAX_HAPPINESS = 100;
    static final int MAX_HEALTH = 100;
    public static final String PET_SYSTEM_VERSION = "2.0";

    // Default constructor
    public VirtualPet() {
```

```
        this("Unnamed", new PetSpecies("DefaultSpecies",
DEFAULT_EVOLUTION_STAGES, 1000, "Forest"), 0, 50, 50);
    }

    public VirtualPet(String petName) {
        this(petName, new PetSpecies("DefaultSpecies",
DEFAULT_EVOLUTION_STAGES, 1000, "Forest"), 0, 50, 50);
    }

    public VirtualPet(String petName, PetSpecies species) {
        this(petName, species, 0, 50, 50);
    }

    public VirtualPet(String petName, PetSpecies species, int age, int
happiness, int health) {
        validateStat(happiness);
        validateStat(health);
        this.petId = generatePetId();
        this.species = species;
        this.birthTimestamp = System.currentTimeMillis();
        this.petName = petName;
        this.age = age;
        this.happiness = happiness;
        this.health = health;
    }

    // JavaBean Getters/Setters
    public String getPetId() { return petId; }
    public PetSpecies getSpecies() { return species; }
    public long getBirthTimestamp() { return birthTimestamp; }
    public String getPetName() { return petName; }
    public void setPetName(String petName) { this.petName = petName; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public int getHappiness() { return happiness; }
    public void setHappiness(int happiness) { validateStat(happiness);
this.happiness = happiness; }
    public int getHealth() { return health; }
    public void setHealth(int health) { validateStat(health); this.health
= health; }
```

```java
public void feedPet(String foodType) {
    modifyHealth(calculateFoodBonus(foodType));
}

public void playWithPet(String gameType) {
    modifyHappiness(calculateGameEffect(gameType));
}

protected int calculateFoodBonus(String foodType) {
    return foodType.equals("Fruit") ? 10 : 5;
}

protected int calculateGameEffect(String gameType) {
    return gameType.equals("Fetch") ? 15 : 8;
}

private void modifyHappiness(int delta) {
    happiness = Math.min(MAX_HAPPINESS, happiness + delta);
    checkEvolution();
}

private void modifyHealth(int delta) {
    health = Math.min(MAX_HEALTH, health + delta);
}

private void updateEvolutionStage() {
    String[] stages = species.getEvolutionStages();
    int currentStageIndex = age / 25; // Evolution every 25 age units

    if (currentStageIndex >= stages.length) {
        currentStageIndex = stages.length - 1; // Max stage
    }

    String newStage = stages[currentStageIndex];
    System.out.println(petName + " is now in stage: " + newStage);
}

public String getInternalState() {
```

```
        return "Pet[" + petId + "] Name: " + petName + ", Age: " + age +
", Health: " + health + ", Happiness: " + happiness;
    }

    private void validateStat(int stat) {
        if (stat < 0 || stat > 100) throw new
IllegalArgumentException("Stat out of range");
    }

    private String generatePetId() {
        return UUID.randomUUID().toString();
    }

    private void checkEvolution() {
        // Check if pet should evolve based on age and happiness
        if (happiness >= 80 && age % 25 == 0 && age > 0) {
            updateEvolutionStage();
            // Bonus stats for evolution
            health = Math.min(MAX_HEALTH, health + 10);
            System.out.println(petName + " gained health from
evolution!");
        }
    }

    @Override
    public String toString() {
        return "VirtualPet{" + "petName='" + petName + '\'' + ", species="
+ species + '}';
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof VirtualPet)) return false;
        VirtualPet other = (VirtualPet) obj;
        return petId.equals(other.petId);
    }

    @Override
    public int hashCode() {
        return petId.hashCode();
```

```
    }
}
```

```
package main;

import model.*;

public class Main {
    public static void main(String[] args) {
        System.out.println("=== VIRTUAL PET SYSTEM DEMO ===\n");

        // Create different types of pets
        PetSpecies dragonSpecies = new PetSpecies(
            "Dragon",
            new String[]{"Egg", "Hatchling", "Wyrmling", "Adult",
"Ancient"},
            5000,
            "Volcano"
        );

        // Test VirtualPet
        System.out.println("1. Basic Virtual Pet:");
        VirtualPet myPet = new VirtualPet("Buddy", dragonSpecies);
        System.out.println(myPet);
        myPet.feedPet("Fruit");
        myPet.playWithPet("Fetch");
        System.out.println(myPet.getInternalState());
        System.out.println();

        // Test DragonPet
        System.out.println("2. Dragon Pet:");
        DragonPet dragon = new DragonPet("Draco", "Fire Dragon", "Fire
Breath");
        System.out.println(dragon);
        dragon.feedPet("Meat");
        dragon.breatheFire();
        dragon.playWithPet("Treasure Hunt");
        dragon.setAge(100);
```

```
        dragon.learnToFly();
        dragon.playWithPet("Flying");
        System.out.println(dragon.getInternalState());
        System.out.println();

        // Test RobotPet
        System.out.println("3. Robot Pet:");
        RobotPet robot = new RobotPet("R2-D2",
            new PetSpecies("Robot", new String[]{"Assembly", "Boot",
"Active", "Advanced"}, 10000, "Laboratory"),
            false, 75);
        System.out.println(robot);
        robot.playWithPet("Logic Games");
        robot.performDiagnostics();
        robot.feedPet("Electricity");
        robot.playWithPet("Logic Games");
        robot.performDiagnostics();
        System.out.println(robot.getInternalState());
        System.out.println();

        // Test evolution
        System.out.println("4. Evolution Test:");
        VirtualPet evolvingPet = new VirtualPet("Evolvo");
        evolvingPet.setHappiness(85);
        evolvingPet.setAge(25); // Should trigger evolution
        evolvingPet.playWithPet("Fetch"); // This will trigger
checkEvolution
        System.out.println(evolvingPet.getInternalState());

        System.out.println("\n=== DEMO COMPLETE ===");
    }
}
```

```
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\VirtualPetSystem> javac -d out src\main\*.java
  src\model\*.java
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\VirtualPetSystem> javac -d out src\main\*.java
  src\model\*.java
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\VirtualPetSystem> java -cp out main.Main
=== VIRTUAL PET SYSTEM DEMO ===

1. Basic Virtual Pet:
VirtualPet{petName='Buddy', species=PetSpecies{speciesName='Dragon', habitat='Volcano'}}
Pet[a4453e51-3f2a-4ef0-888f-ad3c92c377d4] Name: Buddy, Age: 0, Health: 60, Happiness: 65

2. Dragon Pet:
DragonPet{name='Draco', type='Fire Dragon', weapon='Fire Breath', firepower=60, canFly=false}
Draco breathes Fire Breath with power 60!
Draco is hoarding treasure!
Draco has learned to fly!
Draco soars through the skies!
Pet[f03f75fc-081d-488f-a83c-272d037d4bce] Name: Draco, Age: 100, Health: 45, Happiness: 100

3. Robot Pet:
RobotPet{name='R2-D2', batteryLevel=75, needsCharging=false, happiness=50}
Running diagnostics on R2-D2
Battery Level: 70%
Charging needed: false
Health: 50%
R2-D2 is charging. Battery: 95%
Running diagnostics on R2-D2
Battery Level: 90%
Charging needed: false
Health: 50%
Pet[9fa7d10b-0552-4707-82bf-ec742c3dec9e] Name: R2-D2, Age: 0, Health: 50, Happiness: 66

4. Evolution Test:
Initial state: Pet[9b9c444e-6060-4839-bd05-5c2eaa8fc813] Name: Evolvo, Age: 0, Health: 50, Happiness: 50
After happiness boost: Pet[9b9c444e-6060-4839-bd05-5c2eaa8fc813] Name: Evolvo, Age: 0, Health: 50, Happines
s: 85
Evolvo is now in stage: Baby
```

```
4. Evolution Test:
Initial state: Pet[9b9c444e-6060-4839-bd05-5c2eaa8fc813] Name: Evolvo, Age: 0, Health: 50, Happiness: 50
After happiness boost: Pet[9b9c444e-6060-4839-bd05-5c2eaa8fc813] Name: Evolvo, Age: 0, Health: 50, Happines
s: 85
Evolvo is now in stage: Baby
Evolvo gained health from evolution!
Final evolved state: Pet[9b9c444e-6060-4839-bd05-5c2eaa8fc813] Name: Evolvo, Age: 25, Health: 60, Happiness
: 100

5. Polymorphism Test:
Feeding all pets the same food:
Feeding Basic Pet:
Pet[d422b80e-cf11-4486-a472-8f92bd6eabe8] Name: Basic Pet, Age: 0, Health: 55, Happiness: 50

Feeding Flame:
Pet[8f62223d-a7ff-4661-80a4-4b98299439e6] Name: Flame, Age: 0, Health: 55, Happiness: 50

Feeding Robo:
Robots don't eat Meat. Try charging instead!
Pet[155585ca-c36a-4c92-9007-4b126f0b6492] Name: Robo, Age: 0, Health: 50, Happiness: 50


=== DEMO COMPLETE ===
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\VirtualPetSystem>
```

Q2.

```java
import java.util.*;

/**
 * DragonLair class demonstrating dragon types and treasure management
 * Extends MagicalStructure with dragon-specific functionality
```

```java
 */
public class DragonLair extends MagicalStructure {
    // === Final fields ===
    private final String dragonType;

    // === Mutable state ===
    private long treasureValue;
    private int territorialRadius;
    private String currentDragon;
    private Map<String, Integer> treasureInventory;
    private boolean isHoarding;

    // === Constructor variations for different dragon types ===

    // Basic lair constructor
    public DragonLair(String name, String location) {
        super(name, location, 300, false); // High power, initially
inactive
        this.dragonType = "Lesser Dragon";
        this.treasureValue = 1000;
        this.territorialRadius = 5;
        this.currentDragon = "None";
        this.treasureInventory = new HashMap<>();
        this.isHoarding = false;
        initializeBasicTreasure();
    }

    // Fire dragon lair
    public DragonLair(String name, String location, String dragonName) {
        super(name, location, 500, true);
        this.dragonType = "Fire Dragon";
        this.treasureValue = 10000;
        this.territorialRadius = 15;
        setCurrentDragon(dragonName);
        this.treasureInventory = new HashMap<>();
        this.isHoarding = true;
        initializeFireDragonTreasure();
    }

    // Ancient dragon lair with specified type
```

```
    public DragonLair(String name, String location, String dragonName,
String type) {
        super(name, location, 700, true);
        this.dragonType = validateDragonType(type);
        this.treasureValue = calculateInitialTreasure(this.dragonType);
        this.territorialRadius = 25;
        setCurrentDragon(dragonName);
        this.treasureInventory = new HashMap<>();
        this.isHoarding = true;
        initializeTreasureByType(this.dragonType);
    }

    // Fully customized lair
    public DragonLair(String name, String location, String dragonName,
String type,
                      long treasureValue, int radius) {
        super(name, location, 600, true);
        this.dragonType = validateDragonType(type);
        setTreasureValue(treasureValue);
        setTerritorialRadius(radius);
        setCurrentDragon(dragonName);
        this.treasureInventory = new HashMap<>();
        this.isHoarding = true;
        initializeTreasureByType(this.dragonType);
    }

    // === Getters and Setters ===
    public String getDragonType() {
        return dragonType;
    }

    public long getTreasureValue() {
        return treasureValue;
    }

    public void setTreasureValue(long treasureValue) {
        if (treasureValue < 0) {
            throw new IllegalArgumentException("Treasure value cannot be
negative");
        }
```

```
        this.treasureValue = treasureValue;
    }

    public int getTerritorialRadius() {
        return territorialRadius;
    }

    public void setTerritorialRadius(int territorialRadius) {
        if (territorialRadius < 1 || territorialRadius > 100) {
            throw new IllegalArgumentException("Territorial radius must be
between 1 and 100");
        }
        this.territorialRadius = territorialRadius;
    }

    public String getCurrentDragon() {
        return currentDragon;
    }

    public void setCurrentDragon(String dragonName) {
        if (dragonName == null || dragonName.trim().isEmpty()) {
            this.currentDragon = "None";
            setActive(false);
            this.isHoarding = false;
        } else {
            this.currentDragon = dragonName.trim();
            setActive(true);
            setCurrentMaintainer(dragonName);
            this.isHoarding = true;
        }
    }

    public Map<String, Integer> getTreasureInventory() {
        return new HashMap<>(treasureInventory); // Defensive copy
    }

    public boolean isHoarding() {
        return isHoarding;
    }
```

```
    public void setHoarding(boolean hoarding) {
        this.isHoarding = hoarding;
    }

    // === Treasure Management ===
    public void addTreasure(String itemType, int quantity, long value) {
        if (itemType == null || itemType.trim().isEmpty()) {
            throw new IllegalArgumentException("Item type cannot be null
or empty");
        }
        if (quantity <= 0 || value <= 0) {
            throw new IllegalArgumentException("Quantity and value must be
positive");
        }

        String trimmedType = itemType.trim();
        treasureInventory.put(trimmedType,
treasureInventory.getOrDefault(trimmedType, 0) + quantity);
        treasureValue += value;

        System.out.println("Added " + quantity + " " + trimmedType + "
worth " + value + " gold to " + getStructureName());

        if (isHoarding) {
            enhanceMagicPower((int)(value / 100)); // More treasure = more
power
        }
    }

    public boolean removeTreasure(String itemType, int quantity) {
        if (itemType == null || quantity <= 0) return false;

        String trimmedType = itemType.trim();
        Integer currentQuantity = treasureInventory.get(trimmedType);

        if (currentQuantity == null || currentQuantity < quantity) {
            System.out.println("Not enough " + trimmedType + " in the
hoard.");
            return false;
        }
```

```
        if (currentQuantity.equals(quantity)) {
            treasureInventory.remove(trimmedType);
        } else {
            treasureInventory.put(trimmedType, currentQuantity -
quantity);
        }

        // Estimate value lost
        long estimatedValue = calculateItemValue(trimmedType) * quantity;
        treasureValue = Math.max(0, treasureValue - estimatedValue);

        System.out.println("Removed " + quantity + " " + trimmedType + "
from " + getStructureName());
        drainMagicPower((int)(estimatedValue / 200)); // Losing treasure
weakens dragon
        return true;
    }

    public void sortTreasure() {
        if (!isActive() || "None".equals(currentDragon)) {
            System.out.println("No dragon available to sort treasure.");
            return;
        }

        System.out.println(currentDragon + " sorts the treasure hoard at "
+ getStructureName());
        enhanceMagicPower(20);
        treasureValue += treasureInventory.size() * 100; // Organization
increases value
    }

    public void displayTreasureHoard() {
        System.out.println("=== Treasure Hoard of " + getStructureName() +
" ===");
        System.out.println("Total Value: " + treasureValue + " gold");
        System.out.println("Dragon: " + currentDragon + " (" + dragonType
+ ")");
        System.out.println("Territory: " + territorialRadius + " km
radius");
```

```
        if (treasureInventory.isEmpty()) {
            System.out.println("The hoard is empty!");
        } else {
            System.out.println("Treasure Contents:");
            for (Map.Entry<String, Integer> entry :
treasureInventory.entrySet()) {
                System.out.println("  " + entry.getKey() + ": " +
entry.getValue());
            }
        }
    }

    public boolean defendHoard(int attackerPower) {
        if (!isActive()) {
            System.out.println("No dragon to defend the hoard!");
            return false;
        }

        int defensePower = getMagicPower() + (int)(treasureValue / 1000) +
territorialRadius;
        System.out.println(currentDragon + " defends with power: " +
defensePower);
        System.out.println("Attacker power: " + attackerPower);

        if (defensePower >= attackerPower) {
            System.out.println(currentDragon + " successfully defends the
hoard!");
            enhanceMagicPower(10); // Victory strengthens the dragon
            return true;
        } else {
            System.out.println("The hoard has been raided!");
            long stolenValue = treasureValue / 4; // Lose 25% of treasure
            treasureValue -= stolenValue;
            drainMagicPower(50);
            System.out.println("Lost " + stolenValue + " gold worth of
treasure!");
            return false;
        }
    }
```

```
    public void expandTerritory(int expansion) {
        if (expansion <= 0) {
            throw new IllegalArgumentException("Expansion must be
positive");
        }

        if (!isActive()) {
            System.out.println("No dragon to expand territory.");
            return;
        }

        int newRadius = Math.min(100, territorialRadius + expansion);
        int actualExpansion = newRadius - territorialRadius;
        setTerritorialRadius(newRadius);

        if (actualExpansion > 0) {
            System.out.println(currentDragon + " expanded territory by " +
actualExpansion + " km");
            enhanceMagicPower(actualExpansion * 5);
        } else {
            System.out.println("Territory already at maximum size!");
        }
    }

    // === Dragon Type Management ===
    private String validateDragonType(String type) {
        if (type == null || type.trim().isEmpty()) {
            return "Unknown Dragon";
        }

        String trimmed = type.trim();
        String[] validTypes = {"Fire Dragon", "Ice Dragon", "Lightning
Dragon", "Earth Dragon",
                               "Shadow Dragon", "Crystal Dragon", "Ancient
Dragon", "Lesser Dragon"};

        for (String validType : validTypes) {
            if (validType.equalsIgnoreCase(trimmed)) {
                return validType;
```

```
            }
        }

        return trimmed; // Allow custom types
    }

    private long calculateInitialTreasure(String type) {
        switch (type) {
            case "Ancient Dragon": return 50000;
            case "Fire Dragon": return 20000;
            case "Ice Dragon": return 18000;
            case "Lightning Dragon": return 22000;
            case "Earth Dragon": return 15000;
            case "Shadow Dragon": return 25000;
            case "Crystal Dragon": return 30000;
            case "Lesser Dragon": return 5000;
            default: return 10000;
        }
    }

    private long calculateItemValue(String itemType) {
        switch (itemType.toLowerCase()) {
            case "gold coins": return 1;
            case "silver": return 10;
            case "gems": return 100;
            case "artifacts": return 1000;
            case "magical items": return 500;
            case "precious metals": return 50;
            default: return 25;
        }
    }

    // === Initialization methods ===
    private void initializeBasicTreasure() {
        treasureInventory.put("Gold Coins", 1000);
        treasureInventory.put("Silver", 100);
    }

    private void initializeFireDragonTreasure() {
        treasureInventory.put("Gold Coins", 5000);
```

```
        treasureInventory.put("Fire Gems", 50);
        treasureInventory.put("Melted Weapons", 200);
        treasureInventory.put("Charred Artifacts", 20);
    }

    private void initializeTreasureByType(String type) {
        switch (type) {
            case "Ice Dragon":
                treasureInventory.put("Ice Crystals", 100);
                treasureInventory.put("Frozen Gems", 75);
                treasureInventory.put("Silver", 1000);
                break;
            case "Lightning Dragon":
                treasureInventory.put("Storm Gems", 80);
                treasureInventory.put("Electrified Metals", 150);
                treasureInventory.put("Lightning Rods", 25);
                break;
            case "Ancient Dragon":
                treasureInventory.put("Ancient Artifacts", 100);
                treasureInventory.put("Legendary Gems", 200);
                treasureInventory.put("Lost Treasures", 50);
                treasureInventory.put("Gold Coins", 10000);
                break;
            default:
                initializeFireDragonTreasure();
        }
    }

    // === Factory methods for specific dragon types ===
    public static DragonLair createYoungDragonLair(String name, String
location, String dragonName) {
        DragonLair lair = new DragonLair(name, location, dragonName);
        lair.setTreasureValue(2000);
        lair.setTerritorialRadius(8);
        return lair;
    }

    public static DragonLair createAncientLair(String name, String
location, String ancientDragon) {
```

```
        DragonLair lair = new DragonLair(name, location, ancientDragon,
"Ancient Dragon");
        lair.expandTerritory(15); // Ancient dragons have large
territories
        lair.addTreasure("Legendary Artifacts", 10, 50000);
        return lair;
    }

    public static DragonLair createElementalLair(String name, String
location, String dragon, String element) {
        String dragonType = element + " Dragon";
        DragonLair lair = new DragonLair(name, location, dragon,
dragonType);

        // Add element-specific treasures
        switch (element.toLowerCase()) {
            case "fire":
                lair.addTreasure("Fire Opals", 20, 5000);
                break;
            case "ice":
                lair.addTreasure("Frost Diamonds", 15, 6000);
                break;
            case "lightning":
                lair.addTreasure("Storm Sapphires", 18, 5500);
                break;
        }

        return lair;
    }

    // === Utility methods ===
    public int getTreasureTypes() {
        return treasureInventory.size();
    }

    public boolean hasTreasureType(String itemType) {
        return treasureInventory.containsKey(itemType);
    }

    public String getHoardStatus() {
```

```
        if (treasureValue >= 100000) return "Legendary Hoard";
        if (treasureValue >= 50000) return "Great Hoard";
        if (treasureValue >= 20000) return "Substantial Hoard";
        if (treasureValue >= 5000) return "Modest Hoard";
        return "Meager Collection";
    }

    public boolean isWealthyDragon() {
        return treasureValue >= 25000 && treasureInventory.size() >= 5;
    }

    // === Override methods ===
    @Override
    public String getStructureInfo() {
        return String.format("DragonLair: %s (%s, Dragon: %s, Treasure: %d
gold, Territory: %d km)",
            getStructureName(), dragonType, currentDragon, treasureValue,
territorialRadius);
    }

    @Override
    public String toString() {
        return "DragonLair{" +
                "name='" + getStructureName() + '\'' +
                ", location='" + getLocation() + '\'' +
                ", type='" + dragonType + '\'' +
                ", dragon='" + currentDragon + '\'' +
                ", treasure=" + treasureValue +
                ", territory=" + territorialRadius + "km" +
                ", hoarding=" + isHoarding +
                ", active=" + isActive() +
                '}';
    }
}
/**
 * EnchantedCastle class demonstrating castle variations and defense
systems
 * Extends MagicalStructure with castle-specific functionality
 */
public class EnchantedCastle extends MagicalStructure {
```

```java
    // === Final fields ===
    private final String castleType;

    // === Mutable state ===
    private int defenseRating;
    private boolean hasDrawbridge;
    private String currentLord;
    private int garrisonSize;

    // === Constructor variations ===

    // Simple fort constructor
    public EnchantedCastle(String name, String location) {
        super(name, location, 150, true);
        this.castleType = "Simple Fort";
        this.defenseRating = 100;
        this.hasDrawbridge = false;
        this.currentLord = "None";
        this.garrisonSize = 10;
    }

    // Royal castle constructor
    public EnchantedCastle(String name, String location, String lord) {
        super(name, location, 300, true);
        this.castleType = "Royal Castle";
        this.defenseRating = 250;
        this.hasDrawbridge = true;
        setCurrentLord(lord);
        this.garrisonSize = 50;
    }

    // Impregnable fortress constructor
    public EnchantedCastle(String name, String location, String lord,
String fortressType) {
        super(name, location, 500, true);
        if (fortressType == null || fortressType.trim().isEmpty()) {
            this.castleType = "Impregnable Fortress";
        } else {
            this.castleType = fortressType.trim();
        }
```

```
        this.defenseRating = 400;
        this.hasDrawbridge = true;
        setCurrentLord(lord);
        this.garrisonSize = 100;
    }

    // Full specification constructor
    public EnchantedCastle(String name, String location, String lord,
String type,
                           int defenseRating, boolean hasDrawbridge, int
garrison) {
        super(name, location, 400, true);
        this.castleType = (type != null && !type.trim().isEmpty()) ?
type.trim() : "Custom Castle";
        setDefenseRating(defenseRating);
        this.hasDrawbridge = hasDrawbridge;
        setCurrentLord(lord);
        setGarrisonSize(garrison);
    }

    // === Getters and Setters ===
    public String getCastleType() {
        return castleType;
    }

    public int getDefenseRating() {
        return defenseRating;
    }

    public void setDefenseRating(int defenseRating) {
        if (defenseRating < 0 || defenseRating > 1000) {
            throw new IllegalArgumentException("Defense rating must be
between 0 and 1000");
        }
        this.defenseRating = defenseRating;
    }

    public boolean hasDrawbridge() {
        return hasDrawbridge;
    }
```

```
    public void setHasDrawbridge(boolean hasDrawbridge) {
        this.hasDrawbridge = hasDrawbridge;
        if (hasDrawbridge) {
            enhanceDefenses(10);
        }
    }

    public String getCurrentLord() {
        return currentLord;
    }

    public void setCurrentLord(String lord) {
        if (lord == null || lord.trim().isEmpty()) {
            this.currentLord = "None";
            setActive(false);
        } else {
            this.currentLord = lord.trim();
            setActive(true);
            setCurrentMaintainer(lord);
        }
    }

    public int getGarrisonSize() {
        return garrisonSize;
    }

    public void setGarrisonSize(int garrisonSize) {
        if (garrisonSize < 0 || garrisonSize > 500) {
            throw new IllegalArgumentException("Garrison size must be
between 0 and 500");
        }
        this.garrisonSize = garrisonSize;
    }

    // === Castle Management ===
    public void raiseDrawbridge() {
        if (!hasDrawbridge) {
            System.out.println(getStructureName() + " has no drawbridge to
raise!");
```

```
            return;
        }
        System.out.println("Drawbridge raised at " + getStructureName() +
" - castle secured!");
        enhanceDefenses(20);
    }

    public void lowerDrawbridge() {
        if (!hasDrawbridge) {
            System.out.println(getStructureName() + " has no drawbridge to
lower!");
            return;
        }
        System.out.println("Drawbridge lowered at " + getStructureName() +
" - castle accessible.");
        reduceDefenses(20);
    }

    public void enhanceDefenses(int enhancement) {
        int newRating = Math.min(1000, defenseRating + enhancement);
        setDefenseRating(newRating);
        System.out.println(getStructureName() + " defenses enhanced to " +
defenseRating);
    }

    public void reduceDefenses(int reduction) {
        int newRating = Math.max(0, defenseRating - reduction);
        setDefenseRating(newRating);
        if (newRating > 0) {
            System.out.println(getStructureName() + " defenses reduced to
" + defenseRating);
        } else {
            System.out.println(getStructureName() + " defenses have been
completely breached!");
        }
    }

    public void trainGarrison() {
        if (garrisonSize == 0) {
```

```
            System.out.println("No garrison to train at " +
getStructureName());
            return;
        }

        System.out.println("Training garrison of " + garrisonSize + " at "
+ getStructureName());
        enhanceDefenses(garrisonSize / 10);
        enhanceMagicPower(5);
    }

    public void recruitSoldiers(int count) {
        if (count <= 0) {
            throw new IllegalArgumentException("Recruitment count must be
positive");
        }

        int newSize = Math.min(500, garrisonSize + count);
        int recruited = newSize - garrisonSize;
        setGarrisonSize(newSize);

        if (recruited > 0) {
            System.out.println("Recruited " + recruited + " soldiers at "
+ getStructureName());
            enhanceDefenses(recruited / 5);
        } else {
            System.out.println("Castle at maximum garrison capacity!");
        }
    }

    public boolean defendAgainstAttack(int attackPower) {
        int totalDefense = defenseRating + (garrisonSize * 2) +
getMagicPower();

        System.out.println(getStructureName() + " defending with total
power: " + totalDefense);
        System.out.println("Attack power: " + attackPower);

        if (totalDefense >= attackPower) {
```

```
            System.out.println(getStructureName() + " successfully
defended!");
            return true;
        } else {
            System.out.println(getStructureName() + " was breached!");
            reduceDefenses(50);
            setGarrisonSize(Math.max(0, garrisonSize - 10));
            return false;
        }
    }

    // === Factory methods for castle types ===
    public static EnchantedCastle createWatchTower(String name, String
location) {
        EnchantedCastle tower = new EnchantedCastle(name, location);
        tower.setDefenseRating(80);
        tower.setGarrisonSize(5);
        return tower;
    }

    public static EnchantedCastle createRoyalPalace(String name, String
location, String king) {
        EnchantedCastle palace = new EnchantedCastle(name, location, king,
"Royal Palace");
        palace.setDefenseRating(350);
        palace.setGarrisonSize(100);
        palace.enhanceMagicPower(200);
        return palace;
    }

    public static EnchantedCastle createMountainFortress(String name,
String location, String commander) {
        EnchantedCastle fortress = new EnchantedCastle(name, location,
commander, "Mountain Fortress");
        fortress.setDefenseRating(500);
        fortress.setGarrisonSize(150);
        fortress.setHasDrawbridge(true);
        return fortress;
    }
```

```
    public static EnchantedCastle createBorderKeep(String name, String
location, String captain) {
        EnchantedCastle keep = new EnchantedCastle(name, location,
captain, "Border Keep");
        keep.setDefenseRating(200);
        keep.setGarrisonSize(75);
        keep.setHasDrawbridge(true);
        return keep;
    }

    // === Utility methods ===
    public boolean isWellDefended() {
        return defenseRating >= 200 && garrisonSize >= 20;
    }

    public int getTotalDefensivePower() {
        return defenseRating + (garrisonSize * 2) + getMagicPower();
    }

    public String getDefenseStatus() {
        if (defenseRating >= 400) return "Impregnable";
        if (defenseRating >= 250) return "Strong";
        if (defenseRating >= 100) return "Moderate";
        if (defenseRating >= 50) return "Weak";
        return "Defenseless";
    }

    // === Override methods ===
    @Override
    public String getStructureInfo() {
        return String.format("EnchantedCastle: %s (%s, Lord: %s, Defense:
%d, Garrison: %d)",
            getStructureName(), castleType, currentLord, defenseRating,
garrisonSize);
    }

    @Override
    public String toString() {
        return "EnchantedCastle{" +
                "name='" + getStructureName() + '\'' +
```

```
                ", location='" + getLocation() + '\'' +
                ", type='" + castleType + '\'' +
                ", lord='" + currentLord + '\'' +
                ", defense=" + defenseRating +
                ", garrison=" + garrisonSize +
                ", drawbridge=" + hasDrawbridge +
                ", active=" + isActive() +
                '}';
    }
}
import java.util.*;

/**
 * Immutable configuration class for Medieval Kingdom Management
 * Demonstrates final class, defensive copying, and factory methods
 */
public final class KingdomConfig {
    // All fields are final for immutability
    private final String kingdomName;
    private final int foundingYear;
    private final String[] allowedStructureTypes;
    private final Map<String, Integer> resourceLimits;

    // Main constructor with full validation
    public KingdomConfig(String kingdomName, int foundingYear,
                         String[] allowedStructureTypes, Map<String,
Integer> resourceLimits) {
        // Validation
        if (kingdomName == null || kingdomName.trim().isEmpty()) {
            throw new IllegalArgumentException("Kingdom name cannot be
null or empty");
        }
        if (foundingYear < 0 || foundingYear >
Calendar.getInstance().get(Calendar.YEAR)) {
            throw new IllegalArgumentException("Invalid founding year");
        }
        if (allowedStructureTypes == null || allowedStructureTypes.length
== 0) {
            throw new IllegalArgumentException("Must have at least one
allowed structure type");
```

```
        }
        if (resourceLimits == null || resourceLimits.isEmpty()) {
            throw new IllegalArgumentException("Resource limits cannot be
null or empty");
        }

        // Defensive copying for immutability
        this.kingdomName = kingdomName.trim();
        this.foundingYear = foundingYear;
        this.allowedStructureTypes = Arrays.copyOf(allowedStructureTypes,
allowedStructureTypes.length);
        this.resourceLimits = new HashMap<>(resourceLimits);

        // Validate structure types
        for (String type : this.allowedStructureTypes) {
            if (type == null || type.trim().isEmpty()) {
                throw new IllegalArgumentException("Structure type cannot
be null or empty");
            }
        }

        // Validate resource limits
        for (Map.Entry<String, Integer> entry :
this.resourceLimits.entrySet()) {
            if (entry.getKey() == null || entry.getValue() == null ||
entry.getValue() < 0) {
                throw new IllegalArgumentException("Invalid resource limit
entry");
            }
        }
    }

    // Getters only (no setters for immutability)
    public String getKingdomName() {
        return kingdomName;
    }

    public int getFoundingYear() {
        return foundingYear;
    }
```

```
    // Return clones for mutable references
    public String[] getAllowedStructureTypes() {
        return Arrays.copyOf(allowedStructureTypes,
allowedStructureTypes.length);
    }

    public Map<String, Integer> getResourceLimits() {
        return new HashMap<>(resourceLimits);
    }

    // Factory method: createDefaultKingdom()
    public static KingdomConfig createDefaultKingdom() {
        String[] defaultStructures = {"WizardTower", "EnchantedCastle",
"MysticLibrary", "DragonLair"};
        Map<String, Integer> defaultResources = new HashMap<>();
        defaultResources.put("Magic", 1000);
        defaultResources.put("Gold", 5000);
        defaultResources.put("Mana", 2000);
        defaultResources.put("Crystals", 500);

        return new KingdomConfig("Avalon", 1200, defaultStructures,
defaultResources);
    }

    // Factory method: createFromTemplate(String type)
    public static KingdomConfig createFromTemplate(String type) {
        if (type == null) {
            throw new IllegalArgumentException("Template type cannot be
null");
        }

        switch (type.toLowerCase()) {
            case "magical":
                return createMagicalKingdom();
            case "defensive":
                return createDefensiveKingdom();
            case "scholarly":
                return createScholarlyKingdom();
            case "dragon":
```

```
                return createDragonKingdom();
            default:
                throw new IllegalArgumentException("Unknown template type:
" + type);
        }
    }

    private static KingdomConfig createMagicalKingdom() {
        String[] structures = {"WizardTower", "MysticLibrary"};
        Map<String, Integer> resources = new HashMap<>();
        resources.put("Magic", 2000);
        resources.put("Mana", 3000);
        resources.put("Scrolls", 1000);

        return new KingdomConfig("Mystrallia", 800, structures,
resources);
    }

    private static KingdomConfig createDefensiveKingdom() {
        String[] structures = {"EnchantedCastle", "WizardTower"};
        Map<String, Integer> resources = new HashMap<>();
        resources.put("Stone", 5000);
        resources.put("Iron", 3000);
        resources.put("Magic", 1000);

        return new KingdomConfig("Fortressia", 1000, structures,
resources);
    }

    private static KingdomConfig createScholarlyKingdom() {
        String[] structures = {"MysticLibrary", "WizardTower"};
        Map<String, Integer> resources = new HashMap<>();
        resources.put("Books", 10000);
        resources.put("Scrolls", 5000);
        resources.put("Knowledge", 3000);

        return new KingdomConfig("Scholaria", 600, structures, resources);
    }

    private static KingdomConfig createDragonKingdom() {
```

```
        String[] structures = {"DragonLair", "EnchantedCastle"};
        Map<String, Integer> resources = new HashMap<>();
        resources.put("Gold", 10000);
        resources.put("Treasure", 5000);
        resources.put("Territory", 2000);

        return new KingdomConfig("Draconum", 1500, structures, resources);
    }

    // Standard methods
    @Override
    public String toString() {
        return "KingdomConfig{" +
                "kingdomName='" + kingdomName + '\'' +
                ", foundingYear=" + foundingYear +
                ", allowedStructureTypes=" +
Arrays.toString(allowedStructureTypes) +
                ", resourceLimits=" + resourceLimits +
                '}';
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof KingdomConfig)) return false;

        KingdomConfig other = (KingdomConfig) obj;
        return foundingYear == other.foundingYear &&
               Objects.equals(kingdomName, other.kingdomName) &&
               Arrays.equals(allowedStructureTypes,
other.allowedStructureTypes) &&
               Objects.equals(resourceLimits, other.resourceLimits);
    }

    @Override
    public int hashCode() {
        int result = Objects.hash(kingdomName, foundingYear,
resourceLimits);
        result = 31 * result + Arrays.hashCode(allowedStructureTypes);
        return result;
```

```
    }
}
import java.util.*;

/**
 * KingdomManager class demonstrating instanceof usage and polymorphic
interactions
 * Manages a collection of different magical structures with type-specific
behavior
 */
public class KingdomManager {
    // === Fields ===
    private final List<Object> structures; // Stores different structure
types
    private final KingdomConfig config;
    private String kingdomName;

    // === Constructor ===
    public KingdomManager(KingdomConfig config) {
        if (config == null) {
            throw new IllegalArgumentException("Kingdom config cannot be
null");
        }
        this.config = config;
        this.structures = new ArrayList<>();
        this.kingdomName = config.getKingdomName();
    }

    // === Structure Management ===
    public boolean addStructure(Object structure) {
        if (structure == null) {
            throw new IllegalArgumentException("Structure cannot be
null");
        }

        String structureType = determineStructureCategory(structure);
        if (!isStructureTypeAllowed(structureType)) {
            System.out.println("Structure type '" + structureType + "' is
not allowed in " + kingdomName);
            return false;
```

```
        }

        structures.add(structure);
        System.out.println("Added " + structureType + " to " +
kingdomName);
        return true;
    }

    public boolean removeStructure(Object structure) {
        boolean removed = structures.remove(structure);
        if (removed) {
            System.out.println("Removed structure from " + kingdomName);
        }
        return removed;
    }

    public List<Object> getAllStructures() {
        return new ArrayList<>(structures); // Defensive copy
    }

    // === instanceof Type Checking Methods ===

    /**
     * Determines the category of a structure using instanceof
     */
    private String determineStructureCategory(Object structure) {
        if (structure instanceof WizardTower) {
            return "WizardTower";
        } else if (structure instanceof EnchantedCastle) {
            return "EnchantedCastle";
        } else if (structure instanceof MysticLibrary) {
            return "MysticLibrary";
        } else if (structure instanceof DragonLair) {
            return "DragonLair";
        } else if (structure instanceof MagicalStructure) {
            return "MagicalStructure";
        } else {
            return "Unknown";
        }
    }
```

```
    /**
     * Checks if two structures can interact using instanceof
     */
    public static boolean canStructuresInteract(Object s1, Object s2) {
        if (s1 == null || s2 == null) return false;

        // Same type interactions
        if (s1.getClass().equals(s2.getClass())) {
            return true;
        }

        // WizardTower can interact with Libraries and Castles
        if (s1 instanceof WizardTower) {
            return s2 instanceof MysticLibrary || s2 instanceof
EnchantedCastle;
        }
        if (s2 instanceof WizardTower) {
            return s1 instanceof MysticLibrary || s1 instanceof
EnchantedCastle;
        }

        // Libraries can share knowledge with Towers
        if (s1 instanceof MysticLibrary && s2 instanceof WizardTower) {
            return true;
        }
        if (s2 instanceof MysticLibrary && s1 instanceof WizardTower) {
            return true;
        }

        // Castles can coordinate with all structures for defense
        if (s1 instanceof EnchantedCastle || s2 instanceof
EnchantedCastle) {
            return true;
        }

        // Dragon lairs are territorial - limited interactions
        if (s1 instanceof DragonLair || s2 instanceof DragonLair) {
            // Dragons only interact with castles (treaties) or other
dragons (rivalry)
```

```
            return (s1 instanceof DragonLair && s2 instanceof
EnchantedCastle) ||
                    (s2 instanceof DragonLair && s1 instanceof
EnchantedCastle) ||
                    (s1 instanceof DragonLair && s2 instanceof DragonLair);
        }

        return false;
    }

    /**
     * Performs a magic battle between structures using instanceof for
different behaviors
     */
    public static String performMagicBattle(Object attacker, Object
defender) {
        if (attacker == null || defender == null) {
            return "Invalid battle participants";
        }

        int attackPower = getStructureBattlePower(attacker, true);
        int defensePower = getStructureBattlePower(defender, false);

        StringBuilder result = new StringBuilder();
        result.append("MAGICAL BATTLE REPORT\n");
        result.append("Attacker:
").append(getStructureName(attacker)).append(" (Power:
").append(attackPower).append(")\n");
        result.append("Defender:
").append(getStructureName(defender)).append(" (Power:
").append(defensePower).append(")\n");

        // Apply type-specific battle modifiers
        int modifier = getBattleModifier(attacker, defender);
        attackPower += modifier;

        if (modifier != 0) {
            result.append("Type advantage modifier:
").append(modifier).append("\n");
        }
```

```
        result.append("Final attack power:
").append(attackPower).append("\n");

        if (attackPower > defensePower) {
            result.append("RESULT: Attacker WINS!");
            applyBattleEffects(attacker, defender, true);
        } else if (defensePower > attackPower) {
            result.append("RESULT: Defender WINS!");
            applyBattleEffects(attacker, defender, false);
        } else {
            result.append("RESULT: DRAW - No decisive winner");
        }

        return result.toString();
    }

    /**
     * Calculates total kingdom power using instanceof to handle different
structure types
     */
    public static int calculateKingdomPower(Object[] structures) {
        if (structures == null || structures.length == 0) {
            return 0;
        }

        int totalPower = 0;
        int wizardTowers = 0, castles = 0, libraries = 0, dragonLairs = 0;

        for (Object structure : structures) {
            if (structure == null) continue;

            // Base power from structure
            totalPower += getStructureBasePower(structure);

            // Count structure types for synergy bonuses
            if (structure instanceof WizardTower) {
                wizardTowers++;
            } else if (structure instanceof EnchantedCastle) {
                castles++;
```

```
            } else if (structure instanceof MysticLibrary) {
                libraries++;
            } else if (structure instanceof DragonLair) {
                dragonLairs++;
            }
        }

        // Apply synergy bonuses
        totalPower += calculateSynergyBonuses(wizardTowers, castles,
libraries, dragonLairs);

        return totalPower;
    }

    // === Helper Methods for instanceof Operations ===

    private static int getStructureBattlePower(Object structure, boolean
isAttacker) {
        if (structure instanceof WizardTower) {
            WizardTower tower = (WizardTower) structure;
            int power = tower.getMagicPower() + (tower.getSpellCount() *
10);
            return isAttacker ? power + 50 : power; // Wizards are better
attackers
        } else if (structure instanceof EnchantedCastle) {
            EnchantedCastle castle = (EnchantedCastle) structure;
            int power = castle.getMagicPower() + castle.getDefenseRating()
+ (castle.getGarrisonSize() * 2);
            return isAttacker ? power : power + 100; // Castles are better
defenders
        } else if (structure instanceof MysticLibrary) {
            MysticLibrary library = (MysticLibrary) structure;
            int power = library.getMagicPower() +
(library.getKnowledgeLevel() / 2) + (library.getBookCount() * 3);
            return power; // Libraries have consistent power
        } else if (structure instanceof DragonLair) {
            DragonLair lair = (DragonLair) structure;
            int power = lair.getMagicPower() +
(int)(lair.getTreasureValue() / 100) + (lair.getTerritorialRadius() * 5);
```

```
            return isAttacker ? power + 200 : power + 50; // Dragons are
fearsome attackers
        } else if (structure instanceof MagicalStructure) {
            MagicalStructure base = (MagicalStructure) structure;
            return base.getMagicPower();
        }
        return 0;
    }

    private static int getBattleModifier(Object attacker, Object defender)
{
        // Type effectiveness system
        if (attacker instanceof WizardTower && defender instanceof
DragonLair) {
            return 100; // Wizards are effective against dragons
        }
        if (attacker instanceof DragonLair && defender instanceof
WizardTower) {
            return -50; // Dragons struggle against wizards
        }
        if (attacker instanceof EnchantedCastle && defender instanceof
WizardTower) {
            return 75; // Military might vs magical power
        }
        if (attacker instanceof DragonLair && defender instanceof
EnchantedCastle) {
            return 150; // Dragons excel at sieges
        }
        if (attacker instanceof MysticLibrary) {
            return -100; // Libraries are not combat-oriented
        }
        return 0;
    }

    private static void applyBattleEffects(Object attacker, Object
defender, boolean attackerWon) {
        if (attackerWon) {
            enhanceStructure(attacker, 20);
            weakenStructure(defender, 30);
        } else {
```

```
            enhanceStructure(defender, 15);
            weakenStructure(attacker, 20);
        }
    }

    private static void enhanceStructure(Object structure, int amount) {
        if (structure instanceof MagicalStructure) {
            MagicalStructure ms = (MagicalStructure) structure;
            ms.enhanceMagicPower(amount);
        }
    }

    private static void weakenStructure(Object structure, int amount) {
        if (structure instanceof MagicalStructure) {
            MagicalStructure ms = (MagicalStructure) structure;
            ms.drainMagicPower(amount);
        }
    }

    private static String getStructureName(Object structure) {
        if (structure instanceof MagicalStructure) {
            MagicalStructure ms = (MagicalStructure) structure;
            return ms.getStructureName();
        }
        return "Unknown Structure";
    }

    private static int getStructureBasePower(Object structure) {
        if (structure instanceof WizardTower) {
            WizardTower tower = (WizardTower) structure;
            return tower.getMagicPower() + (tower.getSpellCount() * 15);
        } else if (structure instanceof EnchantedCastle) {
            EnchantedCastle castle = (EnchantedCastle) structure;
            return castle.getMagicPower() + (castle.getDefenseRating() /
2) + castle.getGarrisonSize();
        } else if (structure instanceof MysticLibrary) {
            MysticLibrary library = (MysticLibrary) structure;
            return library.getMagicPower() + library.getKnowledgeLevel() +
(library.getBookCount() * 2);
        } else if (structure instanceof DragonLair) {
```

```
            DragonLair lair = (DragonLair) structure;
            return lair.getMagicPower() + (int)(lair.getTreasureValue() /
50) + (lair.getTerritorialRadius() * 3);
        } else if (structure instanceof MagicalStructure) {
            MagicalStructure base = (MagicalStructure) structure;
            return base.getMagicPower();
        }
        return 0;
    }

    private static int calculateSynergyBonuses(int towers, int castles,
int libraries, int lairs) {
        int bonus = 0;

        // Tower-Library synergy (magical research)
        bonus += Math.min(towers, libraries) * 50;

        // Castle-Tower synergy (protected magical research)
        bonus += Math.min(castles, towers) * 30;

        // Multiple castles (defensive network)
        if (castles >= 2) bonus += castles * 25;

        // Dragon diversity bonus
        if (lairs >= 2) bonus += lairs * 40;

        // Complete kingdom bonus (all structure types)
        if (towers > 0 && castles > 0 && libraries > 0 && lairs > 0) {
            bonus += 200;
        }

        return bonus;
    }

    // === Kingdom Management Methods ===

    public void displayKingdomStatus() {
        System.out.println("=== KINGDOM STATUS: " + kingdomName + " ===");
        System.out.println("Founded: " + config.getFoundingYear());
        System.out.println("Total Structures: " + structures.size());
```

```
        Map<String, Integer> structureCount = new HashMap<>();
        for (Object structure : structures) {
            String type = determineStructureCategory(structure);
            structureCount.put(type, structureCount.getOrDefault(type, 0)
+ 1);
        }

        System.out.println("Structure Breakdown:");
        for (Map.Entry<String, Integer> entry : structureCount.entrySet())
{
            System.out.println("  " + entry.getKey() + ": " +
entry.getValue());
        }

        Object[] structureArray = structures.toArray();
        int totalPower = calculateKingdomPower(structureArray);
        System.out.println("Total Kingdom Power: " + totalPower);

        System.out.println("Resource Limits: " +
config.getResourceLimits());
    }

    public List<Object> findStructuresOfType(Class<?> type) {
        List<Object> result = new ArrayList<>();
        for (Object structure : structures) {
            if (type.isInstance(structure)) {
                result.add(structure);
            }
        }
        return result;
    }

    public void performKingdomwideOperation(String operation) {
        System.out.println("Performing kingdom-wide operation: " +
operation);

        for (Object structure : structures) {
            if (structure instanceof WizardTower &&
operation.equals("ENHANCE_MAGIC")) {
```

```
                WizardTower tower = (WizardTower) structure;
                tower.practiceSpells();
            } else if (structure instanceof EnchantedCastle &&
operation.equals("DEFENSE_DRILL")) {
                EnchantedCastle castle = (EnchantedCastle) structure;
                castle.trainGarrison();
            } else if (structure instanceof MysticLibrary &&
operation.equals("ORGANIZE")) {
                MysticLibrary library = (MysticLibrary) structure;
                library.organizeLibrary();
            } else if (structure instanceof DragonLair &&
operation.equals("SORT_TREASURE")) {
                DragonLair lair = (DragonLair) structure;
                lair.sortTreasure();
            }
        }
    }

    private boolean isStructureTypeAllowed(String structureType) {
        String[] allowedTypes = config.getAllowedStructureTypes();
        for (String allowedType : allowedTypes) {
            if (allowedType.equals(structureType)) {
                return true;
            }
        }
        return false;
    }

    // === Getters ===
    public KingdomConfig getConfig() {
        return config;
    }

    public String getKingdomName() {
        return kingdomName;
    }

    public int getStructureCount() {
        return structures.size();
    }
```

```java
}
import java.util.Objects;
import java.util.UUID;

/**
 * Base MagicalStructure class demonstrating constructor chaining,
 * access modifiers, and immutable identity with controlled state
 */
public class MagicalStructure {
    // === Immutable identity ===
    private final String structureId;
    private final long constructionTimestamp;

    // === Immutable properties ===
    private final String structureName;
    private final String location;

    // === Controlled state ===
    private int magicPower;
    private boolean isActive;
    private String currentMaintainer;

    // === Package constants ===
    static final int MIN_MAGIC_POWER = 0;
    static final int MAX_MAGIC_POWER = 1000;

    // === Global constant ===
    public static final String MAGIC_SYSTEM_VERSION = "3.0";

    // === Constructor chaining ===

    // Basic constructor
    public MagicalStructure(String name, String location) {
        this(name, location, 100); // Default power of 100
    }

    // Constructor with power
    public MagicalStructure(String name, String location, int power) {
        this(name, location, power, true); // Default active state
    }
```

```
    // Main constructor - all others chain to this
    public MagicalStructure(String name, String location, int power,
boolean active) {
        // Validation
        if (name == null || name.trim().isEmpty()) {
            throw new IllegalArgumentException("Structure name cannot be
null or empty");
        }
        if (location == null || location.trim().isEmpty()) {
            throw new IllegalArgumentException("Location cannot be null or
empty");
        }
        if (power < MIN_MAGIC_POWER || power > MAX_MAGIC_POWER) {
            throw new IllegalArgumentException("Magic power must be
between " +
                    MIN_MAGIC_POWER + " and " + MAX_MAGIC_POWER);
        }

        // Set immutable identity
        this.structureId = UUID.randomUUID().toString();
        this.constructionTimestamp = System.currentTimeMillis();

        // Set immutable properties
        this.structureName = name.trim();
        this.location = location.trim();

        // Set controlled state
        this.magicPower = power;
        this.isActive = active;
        this.currentMaintainer = "Unknown";
    }

    // === Getters for immutable fields ===
    public String getStructureId() {
        return structureId;
    }

    public long getConstructionTimestamp() {
        return constructionTimestamp;
```

```
    }

    public String getStructureName() {
        return structureName;
    }

    public String getLocation() {
        return location;
    }

    // === JavaBean getters/setters for controlled state ===
    public int getMagicPower() {
        return magicPower;
    }

    public void setMagicPower(int magicPower) {
        if (magicPower < MIN_MAGIC_POWER || magicPower > MAX_MAGIC_POWER)
{
            throw new IllegalArgumentException("Magic power must be
between " +
                    MIN_MAGIC_POWER + " and " + MAX_MAGIC_POWER);
        }
        this.magicPower = magicPower;
    }

    public boolean isActive() {
        return isActive;
    }

    public void setActive(boolean active) {
        this.isActive = active;
    }

    public String getCurrentMaintainer() {
        return currentMaintainer;
    }

    public void setCurrentMaintainer(String currentMaintainer) {
        if (currentMaintainer == null ||
currentMaintainer.trim().isEmpty()) {
```

```
            throw new IllegalArgumentException("Maintainer cannot be null
or empty");
        }
        this.currentMaintainer = currentMaintainer.trim();
    }

    // === Utility methods ===
    public void activateStructure() {
        this.isActive = true;
        System.out.println(structureName + " has been activated!");
    }

    public void deactivateStructure() {
        this.isActive = false;
        System.out.println(structureName + " has been deactivated.");
    }

    public void enhanceMagicPower(int enhancement) {
        int newPower = Math.min(MAX_MAGIC_POWER, magicPower +
enhancement);
        setMagicPower(newPower);
        System.out.println(structureName + " magic power enhanced to " +
magicPower);
    }

    public void drainMagicPower(int drain) {
        int newPower = Math.max(MIN_MAGIC_POWER, magicPower - drain);
        setMagicPower(newPower);
        System.out.println(structureName + " magic power drained to " +
magicPower);
    }

    public long getAge() {
        return System.currentTimeMillis() - constructionTimestamp;
    }

    public String getStructureInfo() {
        return String.format("Structure: %s at %s (Power: %d, Active: %s,
Maintainer: %s)",
```

```
            structureName, location, magicPower, isActive,
currentMaintainer);
    }

    // === Standard methods ===
    @Override
    public String toString() {
        return "MagicalStructure{" +
                "id='" + structureId.substring(0, 8) + "...'" +
                ", name='" + structureName + '\'' +
                ", location='" + location + '\'' +
                ", magicPower=" + magicPower +
                ", isActive=" + isActive +
                ", maintainer='" + currentMaintainer + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof MagicalStructure)) return false;

        MagicalStructure other = (MagicalStructure) obj;
        return Objects.equals(structureId, other.structureId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(structureId);
    }
}
import java.util.*;

/**
 * MysticLibrary class demonstrating knowledge management and book
collections
 * Extends MagicalStructure with library-specific functionality
 */
public class MysticLibrary extends MagicalStructure {
    // === Final fields ===
```

```
    private final Map<String, String> bookCollection;

    // === Mutable state ===
    private int knowledgeLevel;
    private String currentLibrarian;
    private Set<String> availableSubjects;
    private int maxBooks;

    // === Constructor variations ===

    // Few books constructor
    public MysticLibrary(String name, String location) {
        super(name, location, 120, true);
        this.bookCollection = new HashMap<>();
        this.knowledgeLevel = 50;
        this.currentLibrarian = "None";
        this.availableSubjects = new HashSet<>();
        this.maxBooks = 100;
        initializeFewBooks();
    }

    // Moderate collection constructor
    public MysticLibrary(String name, String location, String librarian) {
        super(name, location, 200, true);
        this.bookCollection = new HashMap<>();
        this.knowledgeLevel = 150;
        setCurrentLibrarian(librarian);
        this.availableSubjects = new HashSet<>();
        this.maxBooks = 500;
        initializeModerateCollection();
    }

    // Ancient archives constructor
    public MysticLibrary(String name, String location, String librarian,
int capacity) {
        super(name, location, 400, true);
        this.bookCollection = new HashMap<>();
        this.knowledgeLevel = 300;
        setCurrentLibrarian(librarian);
        this.availableSubjects = new HashSet<>();
```

```
        setMaxBooks(capacity);
        initializeAncientArchives();
    }

    // Custom library constructor
    public MysticLibrary(String name, String location, String librarian,
                         int capacity, Map<String, String> initialBooks) {
        super(name, location, 300, true);
        this.bookCollection = new HashMap<>();
        this.knowledgeLevel = 200;
        setCurrentLibrarian(librarian);
        this.availableSubjects = new HashSet<>();
        setMaxBooks(capacity);

        if (initialBooks != null) {
            for (Map.Entry<String, String> entry :
initialBooks.entrySet()) {
                addBook(entry.getKey(), entry.getValue());
            }
        }
    }

    // === Getters and Setters ===
    public Map<String, String> getBookCollection() {
        return new HashMap<>(bookCollection); // Defensive copy
    }

    public int getKnowledgeLevel() {
        return knowledgeLevel;
    }

    public void setKnowledgeLevel(int knowledgeLevel) {
        if (knowledgeLevel < 0 || knowledgeLevel > 1000) {
            throw new IllegalArgumentException("Knowledge level must be
between 0 and 1000");
        }
        this.knowledgeLevel = knowledgeLevel;
    }

    public String getCurrentLibrarian() {
```

```
        return currentLibrarian;
    }

    public void setCurrentLibrarian(String librarian) {
        if (librarian == null || librarian.trim().isEmpty()) {
            this.currentLibrarian = "None";
            setActive(false);
        } else {
            this.currentLibrarian = librarian.trim();
            setActive(true);
            setCurrentMaintainer(librarian);
        }
    }

    public Set<String> getAvailableSubjects() {
        return new HashSet<>(availableSubjects); // Defensive copy
    }

    public int getMaxBooks() {
        return maxBooks;
    }

    public void setMaxBooks(int maxBooks) {
        if (maxBooks < 10 || maxBooks > 10000) {
            throw new IllegalArgumentException("Max books must be between
10 and 10000");
        }
        this.maxBooks = maxBooks;
    }

    // === Book Management ===
    public boolean addBook(String title, String subject) {
        if (title == null || title.trim().isEmpty()) {
            throw new IllegalArgumentException("Book title cannot be null
or empty");
        }
        if (subject == null || subject.trim().isEmpty()) {
            throw new IllegalArgumentException("Book subject cannot be
null or empty");
        }
```

```
        if (bookCollection.size() >= maxBooks) {
            System.out.println("Library at capacity! Cannot add more
books.");
            return false;
        }

        String trimmedTitle = title.trim();
        String trimmedSubject = subject.trim();

        if (bookCollection.containsKey(trimmedTitle)) {
            System.out.println("Book '" + trimmedTitle + "' already exists
in the library.");
            return false;
        }

        bookCollection.put(trimmedTitle, trimmedSubject);
        availableSubjects.add(trimmedSubject);
        increaseKnowledge(5);
        System.out.println("Added book: '" + trimmedTitle + "' (Subject: "
+ trimmedSubject + ")");
        return true;
    }

    public boolean removeBook(String title) {
        if (title == null) return false;

        String subject = bookCollection.remove(title.trim());
        if (subject != null) {
            System.out.println("Removed book: '" + title + "'");
            decreaseKnowledge(3);

            // Check if this was the last book of this subject
            if (!bookCollection.containsValue(subject)) {
                availableSubjects.remove(subject);
                System.out.println("No more books on " + subject + " -
subject removed.");
            }
            return true;
        }
```

```
        return false;
    }

    public String findBook(String title) {
        return bookCollection.get(title);
    }

    public List<String> findBooksBySubject(String subject) {
        List<String> books = new ArrayList<>();
        for (Map.Entry<String, String> entry : bookCollection.entrySet())
{
            if (entry.getValue().equalsIgnoreCase(subject)) {
                books.add(entry.getKey());
            }
        }
        return books;
    }

    public boolean studyBook(String title) {
        if (!isActive()) {
            System.out.println("Library is closed! Cannot study books.");
            return false;
        }

        String subject = bookCollection.get(title);
        if (subject == null) {
            System.out.println("Book '" + title + "' not found in the
library.");
            return false;
        }

        System.out.println("Studying '" + title + "' on " + subject + " at
" + getStructureName());
        increaseKnowledge(10);
        enhanceMagicPower(5);
        return true;
    }

    public void researchSubject(String subject) {
        if (!isActive()) {
```

```
            System.out.println("Library is closed! Cannot conduct
research.");
            return;
        }

        if (!availableSubjects.contains(subject)) {
            System.out.println("No books available on " + subject);
            return;
        }

        List<String> relevantBooks = findBooksBySubject(subject);
        System.out.println("Researching " + subject + " using " +
relevantBooks.size() + " books");
        increaseKnowledge(relevantBooks.size() * 5);
        enhanceMagicPower(relevantBooks.size() * 2);
    }

    // === Knowledge Management ===
    private void increaseKnowledge(int amount) {
        int newLevel = Math.min(1000, knowledgeLevel + amount);
        setKnowledgeLevel(newLevel);
    }

    private void decreaseKnowledge(int amount) {
        int newLevel = Math.max(0, knowledgeLevel - amount);
        setKnowledgeLevel(newLevel);
    }

    public void organizeLibrary() {
        if ("None".equals(currentLibrarian)) {
            System.out.println("No librarian available to organize the
library.");
            return;
        }

        System.out.println(currentLibrarian + " organizes " +
getStructureName());
        increaseKnowledge(bookCollection.size() / 10);
        enhanceMagicPower(10);
    }
```

```
    public void catalogBooks() {
        System.out.println("=== Library Catalog for " + getStructureName()
+ " ===");
        Map<String, List<String>> subjectGroups = new HashMap<>();

        for (Map.Entry<String, String> entry : bookCollection.entrySet())
{
            String subject = entry.getValue();
            subjectGroups.computeIfAbsent(subject, k -> new
ArrayList<>()).add(entry.getKey());
        }

        for (Map.Entry<String, List<String>> entry :
subjectGroups.entrySet()) {
            System.out.println(entry.getKey() + " (" +
entry.getValue().size() + " books):");
            for (String book : entry.getValue()) {
                System.out.println("  - " + book);
            }
        }
    }

    // === Factory methods for library types ===
    public static MysticLibrary createScholarLibrary(String name, String
location, String scholar) {
        MysticLibrary library = new MysticLibrary(name, location,
scholar);
        library.addBook("Introduction to Magic", "Magic Theory");
        library.addBook("Basic Alchemy", "Alchemy");
        library.addBook("Herb Gathering", "Herbalism");
        return library;
    }

    public static MysticLibrary createRoyalLibrary(String name, String
location, String royalLibrarian) {
        MysticLibrary library = new MysticLibrary(name, location,
royalLibrarian, 1000);
        initializeRoyalCollection(library);
        return library;
```

```
    }

    public static MysticLibrary createAncientRepository(String name,
String location, String keeper) {
        MysticLibrary library = new MysticLibrary(name, location, keeper,
5000);
        initializeAncientCollection(library);
        library.setKnowledgeLevel(500);
        return library;
    }

    // === Initialization methods ===
    private void initializeFewBooks() {
        addBook("Basic Magic", "Magic Theory");
        addBook("Simple Spells", "Spellcasting");
        addBook("Herb Guide", "Herbalism");
    }

    private void initializeModerateCollection() {
        initializeFewBooks();
        addBook("Intermediate Magic", "Magic Theory");
        addBook("Potion Making", "Alchemy");
        addBook("History of Kingdoms", "History");
        addBook("Dragon Lore", "Mythology");
        addBook("Enchantment Basics", "Enchantment");
    }

    private void initializeAncientArchives() {
        initializeModerateCollection();
        addBook("Advanced Transmutation", "Alchemy");
        addBook("Time Magic", "Chronomancy");
        addBook("Planar Travel", "Planar Studies");
        addBook("Ancient Prophecies", "Divination");
        addBook("Lost Civilizations", "Archaeology");
        addBook("Forbidden Arts", "Dark Magic");
    }

    private static void initializeRoyalCollection(MysticLibrary library) {
        String[] subjects = {"Magic Theory", "History", "Politics",
"Military Strategy", "Economics"};
```

```
        String[][] books = {
            {"Royal Magic Protocols", "Court Wizardry", "State
Enchantments"},
            {"Kingdom Chronicles", "Royal Lineages", "Treaties and
Alliances"},
            {"Diplomacy Arts", "Leadership Principles", "Governance
Methods"},
            {"Castle Defense", "Siege Warfare", "Military Tactics"},
            {"Trade Regulations", "Tax Systems", "Resource Management"}
        };

        for (int i = 0; i < subjects.length; i++) {
            for (String book : books[i]) {
                library.addBook(book, subjects[i]);
            }
        }
    }

    private static void initializeAncientCollection(MysticLibrary library)
{
        initializeRoyalCollection(library);
        String[] ancientSubjects = {"Ancient Magic", "Lost Languages",
"Cosmic Studies", "Artifact Creation"};
        String[][] ancientBooks = {
            {"Primordial Spells", "Creation Magic", "World Shaping"},
            {"Dead Languages", "Ancient Runes", "Forgotten Scripts"},
            {"Star Magic", "Celestial Bodies", "Cosmic Forces"},
            {"Legendary Weapons", "Magic Items", "Power Sources"}
        };

        for (int i = 0; i < ancientSubjects.length; i++) {
            for (String book : ancientBooks[i]) {
                library.addBook(book, ancientSubjects[i]);
            }
        }
    }

    // === Utility methods ===
    public int getBookCount() {
        return bookCollection.size();
```

```
    }

    public int getSubjectCount() {
        return availableSubjects.size();
    }

    public boolean hasBook(String title) {
        return bookCollection.containsKey(title);
    }

    public boolean hasSubject(String subject) {
        return availableSubjects.contains(subject);
    }

    public boolean isAtCapacity() {
        return bookCollection.size() >= maxBooks;
    }

    public String getLibraryStatus() {
        if (knowledgeLevel >= 500) return "Ancient Archive";
        if (knowledgeLevel >= 300) return "Great Library";
        if (knowledgeLevel >= 150) return "Scholarly Library";
        if (knowledgeLevel >= 50) return "Basic Library";
        return "Poor Collection";
    }

    // === Override methods ===
    @Override
    public String getStructureInfo() {
        return String.format("MysticLibrary: %s (Librarian: %s, Books:
%d/%d, Knowledge: %d)",
            getStructureName(), currentLibrarian, bookCollection.size(),
maxBooks, knowledgeLevel);
    }

    @Override
    public String toString() {
        return "MysticLibrary{" +
                "name='" + getStructureName() + '\'' +
                ", location='" + getLocation() + '\'' +
```

```
                ", librarian='" + currentLibrarian + '\'' +
                ", books=" + bookCollection.size() + "/" + maxBooks +
                ", knowledge=" + knowledgeLevel +
                ", subjects=" + availableSubjects.size() +
                ", active=" + isActive() +
                '}';
    }
}
/**
 * Main demonstration class for the Medieval Kingdom Management System
 * Shows all features including access modifiers, immutable objects,
instanceof, and constructor overloading
 */
public class Main {
    public static void main(String[] args) {
        System.out.println("🏰 MEDIEVAL KINGDOM MANAGEMENT SYSTEM DEMO
🏰");
        System.out.println("=" .repeat(60));

        // === 1. IMMUTABLE KINGDOM CONFIG DEMONSTRATION ===
        System.out.println("\n1. IMMUTABLE KINGDOM CONFIG");
        System.out.println("-".repeat(30));

        // Default kingdom
        KingdomConfig defaultKingdom =
KingdomConfig.createDefaultKingdom();
        System.out.println("Default Kingdom: " +
defaultKingdom.getKingdomName());

        // Template kingdoms
        KingdomConfig magicalKingdom =
KingdomConfig.createFromTemplate("magical");
        KingdomConfig defensiveKingdom =
KingdomConfig.createFromTemplate("defensive");
        System.out.println("Magical Kingdom: " +
magicalKingdom.getKingdomName());
        System.out.println("Defensive Kingdom: " +
defensiveKingdom.getKingdomName());

        // === 2. CONSTRUCTOR CHAINING DEMONSTRATION ===
```

```
        System.out.println("\n2. CONSTRUCTOR CHAINING EXAMPLES");
        System.out.println("-".repeat(35));

        // MagicalStructure chaining
        MagicalStructure basic = new MagicalStructure("Basic Tower",
"Forest");
        MagicalStructure withPower = new MagicalStructure("Power Tower",
"Mountain", 500);
        MagicalStructure complete = new MagicalStructure("Complete Tower",
"Castle", 800, true);

        System.out.println("Basic constructor: " +
basic.getStructureInfo());
        System.out.println("With power: " + withPower.getStructureInfo());
        System.out.println("Complete: " + complete.getStructureInfo());

        // === 3. WIZARD TOWER VARIATIONS ===
        System.out.println("\n3. WIZARD TOWER CONSTRUCTOR VARIATIONS");
        System.out.println("-".repeat(40));

        WizardTower emptyTower = new WizardTower("Empty Spire",
"Wasteland");
        WizardTower apprenticeTower =
WizardTower.createApprenticeTower("Learning Hall", "Academy", "Young
Mage");
        WizardTower battleTower = WizardTower.createBattleMageTower("War
Spire", "Battlefield", "Battle Mage");
        WizardTower archmageTower = WizardTower.createArchmageTower("Grand
Spire", "Capital", "Archmage Supreme");

        System.out.println("Empty Tower: " +
emptyTower.getStructureInfo());
        System.out.println("Apprentice Tower: " +
apprenticeTower.getStructureInfo());
        System.out.println("Battle Tower: " +
battleTower.getStructureInfo());
        System.out.println("Archmage Tower: " +
archmageTower.getStructureInfo());

        // === 4. CASTLE VARIATIONS ===
```

```
        System.out.println("\n4. ENCHANTED CASTLE VARIATIONS");
        System.out.println("-".repeat(35));

        EnchantedCastle simpleFort = new EnchantedCastle("Border Post",
"North Border");
        EnchantedCastle royalCastle = new EnchantedCastle("Royal Palace",
"Capital", "King Arthur");
        EnchantedCastle mountainFortress =
EnchantedCastle.createMountainFortress("Iron Hold", "High Peak", "General
Stone");

        System.out.println("Simple Fort: " +
simpleFort.getStructureInfo());
        System.out.println("Royal Castle: " +
royalCastle.getStructureInfo());
        System.out.println("Mountain Fortress: " +
mountainFortress.getStructureInfo());

        // === 5. LIBRARY COLLECTIONS ===
        System.out.println("\n5. MYSTIC LIBRARY COLLECTIONS");
        System.out.println("-".repeat(32));

        MysticLibrary basicLibrary = new MysticLibrary("Village Library",
"Small Town");
        MysticLibrary royalLibrary =
MysticLibrary.createRoyalLibrary("Royal Archives", "Palace", "Royal
Librarian");
        MysticLibrary ancientRepository =
MysticLibrary.createAncientRepository("Ancient Vault", "Lost City",
"Eternal Keeper");

        System.out.println("Basic Library: " +
basicLibrary.getStructureInfo());
        System.out.println("Royal Library: " +
royalLibrary.getStructureInfo());
        System.out.println("Ancient Repository: " +
ancientRepository.getStructureInfo());

        // === 6. DRAGON LAIR TYPES ===
        System.out.println("\n6. DRAGON LAIR TYPES");
```

```
        System.out.println("-".repeat(22));

        DragonLair basicLair = new DragonLair("Abandoned Cave", "Dark
Forest");
        DragonLair fireLair = new DragonLair("Volcano Lair", "Fire
Mountain", "Flameheart");
        DragonLair ancientLair = DragonLair.createAncientLair("Ancient
Cavern", "Primordial Peak", "Worldshaker");
        DragonLair iceLair = DragonLair.createElementalLair("Frost
Cavern", "Frozen Wastes", "Iceclaw", "Ice");

        System.out.println("Basic Lair: " + basicLair.getStructureInfo());
        System.out.println("Fire Lair: " + fireLair.getStructureInfo());
        System.out.println("Ancient Lair: " +
ancientLair.getStructureInfo());
        System.out.println("Ice Lair: " + iceLair.getStructureInfo());

        // === 7. KINGDOM MANAGER WITH INSTANCEOF ===
        System.out.println("\n7. KINGDOM MANAGER DEMONSTRATION");
        System.out.println("-".repeat(35));

        KingdomManager kingdom = new KingdomManager(defaultKingdom);

        // Add various structures
        kingdom.addStructure(archmageTower);
        kingdom.addStructure(royalCastle);
        kingdom.addStructure(royalLibrary);
        kingdom.addStructure(fireLair);

        kingdom.displayKingdomStatus();

        // === 8. STRUCTURE INTERACTIONS (instanceof) ===
        System.out.println("\n8. STRUCTURE INTERACTION TESTING");
        System.out.println("-".repeat(35));

        System.out.println("Can Wizard Tower interact with Library? " +
            KingdomManager.canStructuresInteract(archmageTower,
royalLibrary));
        System.out.println("Can Dragon Lair interact with Castle? " +
            KingdomManager.canStructuresInteract(fireLair, royalCastle));
```

```
        System.out.println("Can Library interact with Dragon Lair? " +
            KingdomManager.canStructuresInteract(royalLibrary, fireLair));

        // === 9. MAGIC BATTLE SYSTEM ===
        System.out.println("\n9. MAGICAL BATTLE DEMONSTRATIONS");
        System.out.println("-".repeat(35));

        String battle1 = KingdomManager.performMagicBattle(archmageTower,
fireLair);
        System.out.println(battle1);
        System.out.println();

        String battle2 =
KingdomManager.performMagicBattle(mountainFortress, battleTower);
        System.out.println(battle2);
        System.out.println();

        // === 10. KINGDOM POWER CALCULATION ===
        System.out.println("\n10. KINGDOM POWER CALCULATION");
        System.out.println("-".repeat(32));

        Object[] allStructures = {archmageTower, royalCastle,
royalLibrary, fireLair, mountainFortress};
        int totalPower =
KingdomManager.calculateKingdomPower(allStructures);
        System.out.println("Total Kingdom Power: " + totalPower);

        // === 11. SPECIALIZED BEHAVIORS ===
        System.out.println("\n11. SPECIALIZED STRUCTURE BEHAVIORS");
        System.out.println("-".repeat(38));

        // Wizard Tower behaviors
        System.out.println("=== Wizard Tower Behaviors ===");
        archmageTower.castSpell("Meteor");
        archmageTower.learnSpell("Ultimate Power");
        archmageTower.practiceSpells();

        System.out.println();

        // Castle behaviors
```

```
        System.out.println("=== Castle Behaviors ===");
        royalCastle.raiseDrawbridge();
        royalCastle.trainGarrison();
        royalCastle.defendAgainstAttack(500);

        System.out.println();

        // Library behaviors
        System.out.println("=== Library Behaviors ===");
        royalLibrary.studyBook("Royal Magic Protocols");
        royalLibrary.researchSubject("Magic Theory");
        royalLibrary.organizeLibrary();

        System.out.println();

        // Dragon Lair behaviors
        System.out.println("=== Dragon Lair Behaviors ===");
        fireLair.addTreasure("Fire Rubies", 10, 5000);
        fireLair.sortTreasure();
        fireLair.defendHoard(800);

        // === 12. KINGDOM-WIDE OPERATIONS ===
        System.out.println("\n12. KINGDOM-WIDE OPERATIONS");
        System.out.println("-".repeat(30));

        kingdom.performKingdomwideOperation("ENHANCE_MAGIC");
        System.out.println();
        kingdom.performKingdomwideOperation("DEFENSE_DRILL");
        System.out.println();

        // === 13. FINAL KINGDOM STATUS ===
        System.out.println("\n13. FINAL KINGDOM STATUS");
        System.out.println("-".repeat(27));
        kingdom.displayKingdomStatus();

        // === 14. IMMUTABILITY DEMONSTRATION ===
        System.out.println("\n14. IMMUTABILITY DEMONSTRATION");
        System.out.println("-".repeat(33));

        // Show that KingdomConfig is truly immutable
```

```
        String[] originalTypes =
defaultKingdom.getAllowedStructureTypes();
        originalTypes[0] = "ModifiedType"; // Try to modify the returned
array

        String[] stillOriginal =
defaultKingdom.getAllowedStructureTypes();
        System.out.println("Original config unchanged: " +
stillOriginal[0]); // Still "WizardTower"

        System.out.println("\n🎉 MEDIEVAL KINGDOM MANAGEMENT SYSTEM DEMO
COMPLETE! 🎉");
    }
}
```

```
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\MedievalKingdom> java Main
? MEDIEVAL KINGDOM MANAGEMENT SYSTEM DEMO ?
============================================================

1. IMMUTABLE KINGDOM CONFIG
----------------------------
Default Kingdom: Avalon
Magical Kingdom: Mystrallia
Defensive Kingdom: Fortressia

2. CONSTRUCTOR CHAINING EXAMPLES
---------------------------------
Basic constructor: Structure: Basic Tower at Forest (Power: 100, Active: true, Maintainer: Unknown)
With power: Structure: Power Tower at Mountain (Power: 500, Active: true, Maintainer: Unknown)
Complete: Structure: Complete Tower at Castle (Power: 800, Active: true, Maintainer: Unknown)

3. WIZARD TOWER CONSTRUCTOR VARIATIONS
---------------------------------------
Spell 'Light' already known.
Spell 'Detect Magic' already known.
Learned new spell: Fireball
War Spire magic power enhanced to 410
Learned new spell: Lightning Bolt
War Spire magic power enhanced to 420
Learned new spell: Magic Missile
War Spire magic power enhanced to 430
Learned new spell: Shield
War Spire magic power enhanced to 440
Learned new spell: Heal
War Spire magic power enhanced to 450
Learned new spell: Meteor
Grand Spire magic power enhanced to 810
Learned new spell: Time Stop
Grand Spire magic power enhanced to 820
Learned new spell: Wish
Grand Spire magic power enhanced to 830
Learned new spell: Gate
Grand Spire magic power enhanced to 840
Learned new spell: Resurrection
```

3. WIZARD TOWER CONSTRUCTOR VARIATIONS

Spell 'Light' already known.
Spell 'Detect Magic' already known.
Learned new spell: Fireball
War Spire magic power enhanced to 410
Learned new spell: Lightning Bolt
War Spire magic power enhanced to 420
Learned new spell: Magic Missile
War Spire magic power enhanced to 430
Learned new spell: Shield
War Spire magic power enhanced to 440
Learned new spell: Heal
War Spire magic power enhanced to 450

Learned new spell: Meteor
Grand Spire magic power enhanced to 810
Learned new spell: Time Stop
Grand Spire magic power enhanced to 820
Learned new spell: Wish
Grand Spire magic power enhanced to 830
Learned new spell: Gate
Grand Spire magic power enhanced to 840
Learned new spell: Resurrection
Grand Spire magic power enhanced to 850
Learned new spell: Disintegrate
Grand Spire magic power enhanced to 860
Learned new spell: Power Word Kill
Grand Spire magic power enhanced to 870
Learned new spell: Mass Heal
Grand Spire magic power enhanced to 880
Empty Tower: WizardTower: Empty Spire (Wizard: None, Spells: 0/10, Power: 200)
Apprentice Tower: WizardTower: Learning Hall (Wizard: Young Mage, Spells: 3/15, Power: 300)
Battle Tower: WizardTower: War Spire (Wizard: Battle Mage, Spells: 5/20, Power: 450)
Archmage Tower: WizardTower: Grand Spire (Wizard: Archmage Supreme, Spells: 8/30, Power: 880)

4. ENCHANTED CASTLE VARIATIONS

Iron Hold defenses enhanced to 510
Simple Fort: EnchantedCastle: Border Post (Simple Fort, Lord: None, Defense: 100, Garrison: 10)
Royal Castle: EnchantedCastle: Royal Palace (Royal Castle, Lord: King Arthur, Defense: 250, Garrison: 50)
Mountain Fortress: EnchantedCastle: Iron Hold (Mountain Fortress, Lord: General Stone, Defense: 510, Garrison: 150)

5. MYSTIC LIBRARY COLLECTIONS

Added book: 'Basic Magic' (Subject: Magic Theory)
Added book: 'Simple Spells' (Subject: Spellcasting)
Added book: 'Herb Guide' (Subject: Herbalism)
Added book: 'Basic Magic' (Subject: Magic Theory)
Added book: 'Simple Spells' (Subject: Spellcasting)
Added book: 'Herb Guide' (Subject: Herbalism)
Added book: 'Intermediate Magic' (Subject: Magic Theory)
Added book: 'Potion Making' (Subject: Alchemy)
Added book: 'History of Kingdoms' (Subject: History)
Added book: 'Dragon Lore' (Subject: Mythology)

Added book: 'Enchantment Basics' (Subject: Enchantment)
Added book: 'Advanced Transmutation' (Subject: Alchemy)
Added book: 'Time Magic' (Subject: Chronomancy)
Added book: 'Planar Travel' (Subject: Planar Studies)
Added book: 'Ancient Prophecies' (Subject: Divination)
Added book: 'Lost Civilizations' (Subject: Archaeology)
Added book: 'Forbidden Arts' (Subject: Dark Magic)
Added book: 'Royal Magic Protocols' (Subject: Magic Theory)
Added book: 'Court Wizardry' (Subject: Magic Theory)
Added book: 'State Enchantments' (Subject: Magic Theory)
Added book: 'Kingdom Chronicles' (Subject: History)
Added book: 'Royal Lineages' (Subject: History)
Added book: 'Treaties and Alliances' (Subject: History)
Added book: 'Diplomacy Arts' (Subject: Politics)
Added book: 'Leadership Principles' (Subject: Politics)
Added book: 'Governance Methods' (Subject: Politics)
Added book: 'Castle Defense' (Subject: Military Strategy)
Added book: 'Siege Warfare' (Subject: Military Strategy)
Added book: 'Military Tactics' (Subject: Military Strategy)
Added book: 'Trade Regulations' (Subject: Economics)
Added book: 'Tax Systems' (Subject: Economics)
Added book: 'Resource Management' (Subject: Economics)
Added book: 'Basic Magic' (Subject: Magic Theory)
Added book: 'Simple Spells' (Subject: Spellcasting)
Added book: 'Herb Guide' (Subject: Herbalism)
Added book: 'Intermediate Magic' (Subject: Magic Theory)
Added book: 'Potion Making' (Subject: Alchemy)
Added book: 'History of Kingdoms' (Subject: History)
Added book: 'Dragon Lore' (Subject: Mythology)
Added book: 'Enchantment Basics' (Subject: Enchantment)
Added book: 'Advanced Transmutation' (Subject: Alchemy)
Added book: 'Time Magic' (Subject: Chronomancy)
Added book: 'Planar Travel' (Subject: Planar Studies)
Added book: 'Ancient Prophecies' (Subject: Divination)
Added book: 'Lost Civilizations' (Subject: Archaeology)
Added book: 'Forbidden Arts' (Subject: Dark Magic)
Added book: 'Royal Magic Protocols' (Subject: Magic Theory)
Added book: 'Court Wizardry' (Subject: Magic Theory)
Added book: 'State Enchantments' (Subject: Magic Theory)
Added book: 'Kingdom Chronicles' (Subject: History)
Added book: 'Royal Lineages' (Subject: History)
Added book: 'Treaties and Alliances' (Subject: History)
Added book: 'Diplomacy Arts' (Subject: Politics)
Added book: 'Leadership Principles' (Subject: Politics)

Added book: 'Governance Methods' (Subject: Politics)
Added book: 'Castle Defense' (Subject: Military Strategy)
Added book: 'Siege Warfare' (Subject: Military Strategy)
Added book: 'Military Tactics' (Subject: Military Strategy)
Added book: 'Trade Regulations' (Subject: Economics)
Added book: 'Tax Systems' (Subject: Economics)
Added book: 'Resource Management' (Subject: Economics)
Added book: 'Primordial Spells' (Subject: Ancient Magic)
Added book: 'Creation Magic' (Subject: Ancient Magic)
Added book: 'World Shaping' (Subject: Ancient Magic)
Added book: 'Dead Languages' (Subject: Lost Languages)
Added book: 'Ancient Runes' (Subject: Lost Languages)
Added book: 'Forgotten Scripts' (Subject: Lost Languages)
Added book: 'Star Magic' (Subject: Cosmic Studies)
Added book: 'Celestial Bodies' (Subject: Cosmic Studies)
Added book: 'Cosmic Forces' (Subject: Cosmic Studies)
Added book: 'Legendary Weapons' (Subject: Artifact Creation)
Added book: 'Magic Items' (Subject: Artifact Creation)
Added book: 'Power Sources' (Subject: Artifact Creation)
Basic Library: MysticLibrary: Village Library (Librarian: None, Books: 3/100, Knowledge: 65)
Royal Library: MysticLibrary: Royal Archives (Librarian: Royal Librarian, Books: 29/1000, Knowledge: 445)
Ancient Repository: MysticLibrary: Ancient Vault (Librarian: Eternal Keeper, Books: 41/5000, Knowledge: 500)

6. DRAGON LAIR TYPES

Worldshaker expanded territory by 15 km
Ancient Cavern magic power enhanced to 775
Added 10 Legendary Artifacts worth 50000 gold to Ancient Cavern
Ancient Cavern magic power enhanced to 1000
Added 15 Frost Diamonds worth 6000 gold to Frost Cavern
Frost Cavern magic power enhanced to 760
Basic Lair: DragonLair: Abandoned Cave (Lesser Dragon, Dragon: None, Treasure: 1000 gold, Territory: 5 km)
Fire Lair: DragonLair: Volcano Lair (Fire Dragon, Dragon: Flameheart, Treasure: 10000 gold, Territory: 15 km)
Ancient Lair: DragonLair: Ancient Cavern (Ancient Dragon, Dragon: Worldshaker, Treasure: 100000 gold, Territory: 40 km)
Ice Lair: DragonLair: Frost Cavern (Ice Dragon, Dragon: Iceclaw, Treasure: 24000 gold, Territory: 25 km)

7. KINGDOM MANAGER DEMONSTRATION

Added WizardTower to Avalon
Added EnchantedCastle to Avalon
Added MysticLibrary to Avalon
Added DragonLair to Avalon
=== KINGDOM STATUS: Avalon ===
Founded: 1200
Total Structures: 4
Structure Breakdown:
  EnchantedCastle: 1
  WizardTower: 1
  DragonLair: 1
  MysticLibrary: 1
Total Kingdom Power: 3403
Resource Limits: {Gold=5000, Mana=2000, Magic=1000, Crystals=500}

8. STRUCTURE INTERACTION TESTING

Can Wizard Tower interact with Library? true
Can Dragon Lair interact with Castle? true
Can Library interact with Dragon Lair? false

9. MAGICAL BATTLE DEMONSTRATIONS

Grand Spire magic power enhanced to 900
Volcano Lair magic power drained to 470
MAGICAL BATTLE REPORT
Attacker: Grand Spire (Power: 1010)
Defender: Volcano Lair (Power: 725)
Type advantage modifier: 100
Final attack power: 1110
RESULT: Attacker WINS!

Iron Hold magic power enhanced to 520
War Spire magic power drained to 420
MAGICAL BATTLE REPORT
Attacker: Iron Hold (Power: 1310)
Defender: War Spire (Power: 500)
Type advantage modifier: 75
Final attack power: 1385
RESULT: Attacker WINS!

10. KINGDOM POWER CALCULATION

Total Kingdom Power: 4368

11. SPECIALIZED STRUCTURE BEHAVIORS

=== Wizard Tower Behaviors ===
Grand Spire magic power drained to 880
Archmage Supreme casts Meteor from Grand Spire!
Learned new spell: Ultimate Power
Grand Spire magic power enhanced to 890
Archmage Supreme practices spells at Grand Spire
Grand Spire magic power enhanced to 895

=== Castle Behaviors ===
Drawbridge raised at Royal Palace - castle secured!
Royal Palace defenses enhanced to 270
Training garrison of 50 at Royal Palace
Royal Palace defenses enhanced to 275
Royal Palace magic power enhanced to 305
Royal Palace defending with total power: 680
Attack power: 500
Royal Palace successfully defended!

=== Library Behaviors ===
Studying 'Royal Magic Protocols' on Magic Theory at Royal Archives
Royal Archives magic power enhanced to 405
Researching Magic Theory using 5 books
Royal Archives magic power enhanced to 415
Royal Librarian organizes Royal Archives
Royal Archives magic power enhanced to 425

=== Dragon Lair Behaviors ===
Added 10 Fire Rubies worth 5000 gold to Volcano Lair
Volcano Lair magic power enhanced to 520
Flameheart sorts the treasure hoard at Volcano Lair
Volcano Lair magic power enhanced to 540
Flameheart defends with power: 570
Attacker power: 800
The hoard has been raided!
Volcano Lair magic power drained to 490
Lost 3875 gold worth of treasure!

12. KINGDOM-WIDE OPERATIONS

Performing kingdom-wide operation: ENHANCE_MAGIC

Archmage Supreme practices spells at Grand Spire
Grand Spire magic power enhanced to 900

Performing kingdom-wide operation: DEFENSE_DRILL
12. KINGDOM-WIDE OPERATIONS

Performing kingdom-wide operation: ENHANCE_MAGIC
Archmage Supreme practices spells at Grand Spire
Grand Spire magic power enhanced to 900

Performing kingdom-wide operation: DEFENSE_DRILL
Performing kingdom-wide operation: ENHANCE_MAGIC
Archmage Supreme practices spells at Grand Spire
Grand Spire magic power enhanced to 900

Performing kingdom-wide operation: DEFENSE_DRILL

Performing kingdom-wide operation: DEFENSE_DRILL
Performing kingdom-wide operation: DEFENSE_DRILL
Training garrison of 50 at Royal Palace
Royal Palace defenses enhanced to 280
Royal Palace magic power enhanced to 310

13. FINAL KINGDOM STATUS

=== KINGDOM STATUS: Avalon ===
Founded: 1200
Total Structures: 4
Structure Breakdown:
  EnchantedCastle: 1
  WizardTower: 1
  DragonLair: 1
Royal Palace defenses enhanced to 280
Royal Palace magic power enhanced to 310

13. FINAL KINGDOM STATUS

=== KINGDOM STATUS: Avalon ===
Founded: 1200
Total Structures: 4
Structure Breakdown:
  EnchantedCastle: 1

```
  WizardTower: 1
  DragonLair: 1


13. FINAL KINGDOM STATUS
--------------------------
=== KINGDOM STATUS: Avalon ===
Founded: 1200
Total Structures: 4
Structure Breakdown:
  EnchantedCastle: 1
  WizardTower: 1
  DragonLair: 1
=== KINGDOM STATUS: Avalon ===
Founded: 1200
Total Structures: 4
Structure Breakdown:
  EnchantedCastle: 1
  WizardTower: 1
  DragonLair: 1
Structure Breakdown:
  EnchantedCastle: 1
  WizardTower: 1
  DragonLair: 1
  WizardTower: 1
  DragonLair: 1
  DragonLair: 1
  MysticLibrary: 1
Total Kingdom Power: 3547
Resource Limits: {Gold=5000, Mana=2000, Magic=1000, Crystals=500}

14. IMMUTABILITY DEMONSTRATION
---------------------------------
Original config unchanged: WizardTower

? MEDIEVAL KINGDOM MANAGEMENT SYSTEM DEMO COMPLETE! ?
PS E:\JAVA PROGRAMS\steparyansingh\year2\oops\week5\lab-work\MedievalKingdom>
```