**Activity 20 : Advanced Pet Sound Simulation & Liskov Substitution**

📂 **Filename**: **python\_activity20.py**

**🎯 Objective:**  
 In this activity, you will create an advanced pet sound simulation where different types of animals (real and robotic) produce sounds. You will implement the **Liskov Substitution Principle (LSP)** by ensuring that subclasses can be substituted for their parent class without breaking the behavior of the system. You will also extend the program to handle more complex behaviors and interactions between different pet types.

**Instructions:**

1. **Step 1: Design the Base Class**
   * Create a base class Animal with the following methods:
     + make\_sound() – This should be overridden by each subclass. It should simulate the animal's primary sound (e.g., "Woof!" for dogs).
     + get\_energy\_level() – Return the animal's current energy level (a numeric value between 1-100). Each subclass will implement this based on its type.
     + perform\_action() – A generic method that will simulate a default action (e.g., walking, running) for all animals. It must be overridden by each subclass to simulate unique actions.
     + Think about how real pets and robots might behave differently in these areas.
2. **Step 2: Build Your Subclasses**
   * Implement **at least four subclasses**: Dog, Cat, RobotDog, and RobotCat. Each subclass must:
     + Override make\_sound() with a unique sound.
     + Implement the get\_energy\_level() method. For example:
       - Dog: Energy decreases over time after multiple actions (simulate fatigue).
       - RobotDog: Energy is constant (robots don’t get tired).
     + Implement the perform\_action() method to simulate a unique behavior for each pet:
       - Dog: Can perform actions like bark() or run().
       - RobotDog: Can perform actions like charge\_battery().
       - RobotCat: Can perform actions like scan\_environment().
3. **Step 3: Test Substitution**
   * Create a function perform\_sound\_and\_action(animal) that takes an Animal object and:
     + Calls make\_sound() to produce the pet's sound.
     + Calls perform\_action() to simulate the pet’s behavior.
     + Prints the energy level of the pet after each action is performed.
   * This function should work seamlessly with **any subclass** of Animal (real or robotic) without needing modifications. This is a true test of **Liskov Substitution Principle**.
4. **Step 4: Handle Special Cases**
   * Add special behaviors for robots in your system:
     + For example, a RobotDog might need to **recharge** if its energy level drops below 10. This should not apply to regular animals, so ensure you correctly handle this distinction without violating the Liskov Substitution Principle.
   * You can introduce additional subclasses, like a RobotCow or a CyberChicken, but they must integrate smoothly with the existing system.

**Here’s the code template to get started:**

class Animal:

def make\_sound(self):

raise NotImplementedError("Subclasses must implement this method!")

def get\_energy\_level(self):

raise NotImplementedError("Subclasses must implement this method!")

def perform\_action(self):

raise NotImplementedError("Subclasses must implement this method!")

class Dog(Animal):

def make\_sound(self):

print("Woof!")

def get\_energy\_level(self):

return 80 # Example energy level for a dog.

def perform\_action(self):

print("Dog is running!")

class Cat(Animal):

def make\_sound(self):

print("Meow!")

def get\_energy\_level(self):

return 75 # Example energy level for a cat.

def perform\_action(self):

print("Cat is purring!")

class RobotDog(Animal):

def make\_sound(self):

print("Beep-Bop!")

def get\_energy\_level(self):

return 100 # RobotDog has constant energy.

def perform\_action(self):

print("RobotDog is charging!")

class RobotCat(Animal):

def make\_sound(self):

print("Meow... Beep!")

def get\_energy\_level(self):

return 100 # RobotCat has constant energy.

def perform\_action(self):

print("RobotCat is scanning the environment!")

def perform\_sound\_and\_action(animal: Animal):

animal.make\_sound()

animal.perform\_action()

print(f"Energy Level: {animal.get\_energy\_level()}")

# Test cases

dog = Dog()

cat = Cat()

robot\_dog = RobotDog()

robot\_cat = RobotCat()

perform\_sound\_and\_action(dog) # Woof! Dog is running! Energy Level: 80

perform\_sound\_and\_action(cat) # Meow! Cat is purring! Energy Level: 75

perform\_sound\_and\_action(robot\_dog) # Beep-Bop! RobotDog is charging! Energy Level: 100

perform\_sound\_and\_action(robot\_cat) # Meow... Beep! RobotCat is scanning the environment! Energy Level: 100