Part 4: Bad Design Example

```
class Animal {
    String name;
    void makeSound() {
        System.out.println("Generic animal sound");
    }
}

class RobotDog extends Animal {
    int batteryLevel;
    void makeSound(String sound) { // Overloading instead of overriding
        System.out.println(sound);
    }
}
```

Tasks:

1. What are the design flaws here?

   Animal should be abstract ,here all animals are forced to use the generic sound method and thats wrong cuz not all animals has the same sound , and using overloading instead of overriding , attributes are not protected and there are no constructors.

2. Why is using method overloading instead of overriding problematic?

   Overloading creates a new method instead of replacing the parent's method, so if you call makeSound() on an Animal reference pointing to a RobotDog object, it will run the parent method instead of RobotDog's. Polymorphism doesn't work here.

3. How can you fix the inheritance and method structure?

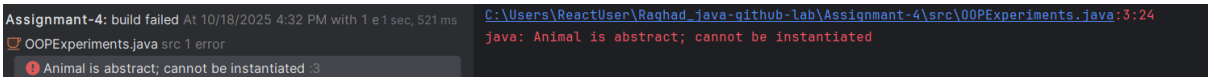
   Make Animal abstract and declare makeSound() as abstract, In RobotDog, using @Override to override the method.

   Add constructors and use super() to initialize the parent fields.

Part 5: Experiments with Type Assignments

Try the following code snippets and observe the results:

```
RobotDog dog = new Animal();          // ❓
```




```
Animal dog2 = new RobotDog();         // ✅

Rechargeable dog3 = new RobotDog();   // ✅ (if Rechargeable is an
interface)
```

Answer these:

1. Which lines compile successfully? Why or why not?

   `RobotDog dog = new Animal();` //Compile error – A parent (Animal) can't be stored in a child (RobotDog) variable.

   `Animal dog2 = new RobotDog();`//works cuz a child object (RobotDog) can be stored in a parent variable.

   `Rechargeable dog3 = new RobotDog();`//works only if RobotDog implements the Rechargeable interface then here it works .

2. What methods can each variable access?

   dog → invalid, doesn't exist.

   dog2 → can access only methods declared in Animal (like eat() and makeSound()), even if the actual object is RobotDog.

   dog3→ can access only methods declared in the Rechargeable interface

3. What happens when you call dog2.makeSound()?

   Java calls RobotDog's overridden makeSound() method. This is runtime polymorphism, even if the dog2 is an animal .

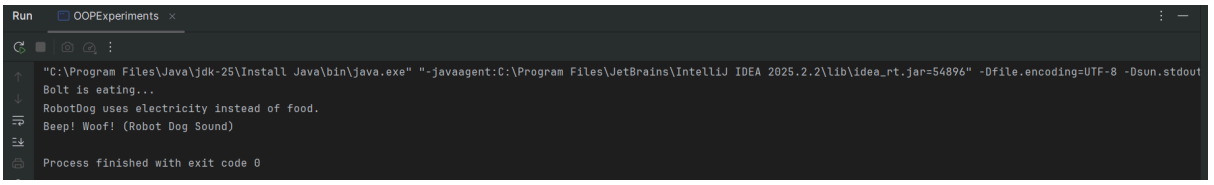
4. How does polymorphism affect method calls?
   Polymorphism lets a parent variable hold a child object and run the child's overridden methods . The variable type decides what methods you can see, but the actual object decides which method runs.

Part 6: Override Behavior Exploration

```
Animal a = new RobotDog("Bolt", 2, 90);

a.eat();

a.makeSound();
```



Questions:

- Which version of eat() is executed — Animal or RobotDog?

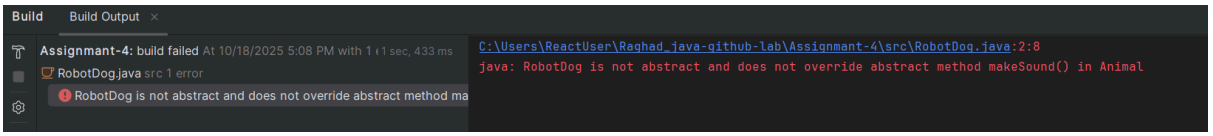
  The RobotDog version of eat() is executed cuz it overrides the parent method.

- Why does Java choose that version at runtime

  Java uses runtime polymorphism. Even if the reference is Animal, the real object is RobotDog.

- What happens if makeSound() is not overridden in RobotDog?

  If it is not overridden, calling a.makeSound() will run the Animal version of the method.



Reflection

Discuss:

1. How does abstraction improve flexibility when adding new types of animals (e.g., `Bird`, `Cat`, `RobotCat`)?

   When we adding a new type of animals we don't have to rewrite the existing code or chang it .

2. What would happen if `Animal` were not abstract?

If Animal is not abstract, we could create generic Animal objects, which don't have a specific sound, This can lead to incorrect behavior and less meaningful code.

3. Why is it better to depend on abstractions (interfaces, abstract classes) rather than concrete implementations?

   makes the code more flexible, easier to maintain, and allows polymorphism. We can change implementations without affecting the rest of the code.