MODULE 3

SECTION 2.3

1.

public abstract class AbstractBankAccount {

protected String accountNumber;

protected double balance;

@Override

public String toString() {

return "Account Number: " + accountNumber + "\nBalance: " + balance;

}

}

public class JavaBank {

private AbstractBankAccount[] myAccounts;

private JTextArea jTextArea;

public JavaBank() {

myAccounts = new AbstractBankAccount[10]; // Example initialization

jTextArea = new JTextArea(); // Initialize the JTextArea

}

private void displayAccountDetails(AbstractBankAccount account) {

jTextArea.setText(account.toString());

}

private void someMethodToDisplayAccount() {

int index = getSelectedAccountIndex();

displayAccountDetails(myAccounts[index]);

}

private int getSelectedAccountIndex() {

return 0; // Example method for getting selected account index

}

public static void main(String[] args) {

JavaBank bank = new JavaBank();

bank.someMethodToDisplayAccount();

}

}

public class SavingsAccount extends AbstractBankAccount {

public SavingsAccount(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

}

public class CheckingAccount extends AbstractBankAccount {

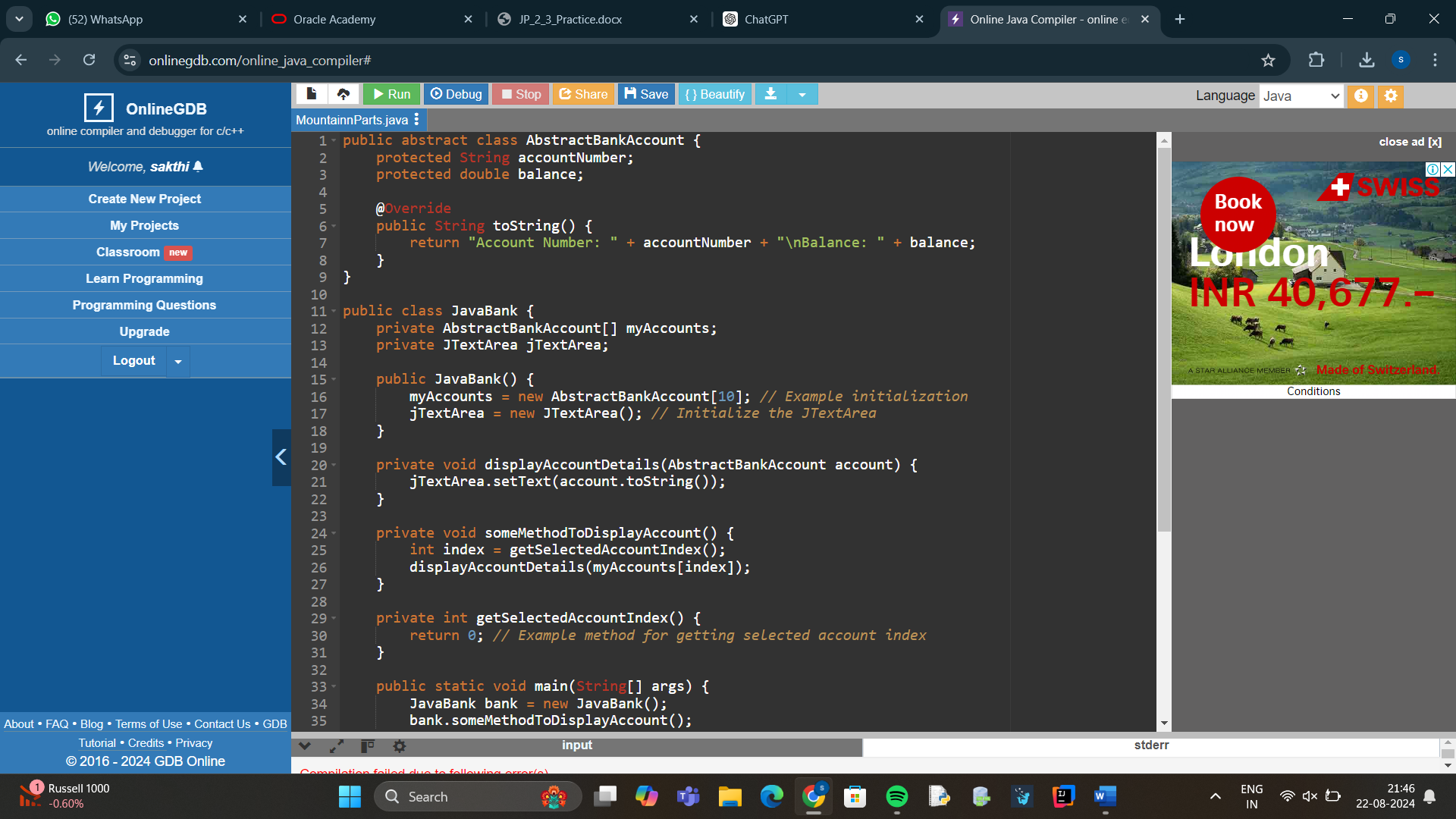
public CheckingAccount(String accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

}



2. One reason to use an abstract class rather than an interface is when you need to provide a common base implementation for some methods while allowing subclasses to provide their own specific implementations.

\*\*Example Reason\*\*:

- \*\*Code Reusability\*\*: Abstract classes allow you to define and provide default behavior for methods that can be shared across multiple subclasses. This avoids code duplication and makes it easier to manage common functionality. For instance, if you have several types of bank accounts that share common behaviors (like account number management and basic balance operations), you can implement these shared methods in an abstract class, while still allowing subclasses to extend or override them as needed. This can simplify your code and reduce redundancy.

3. public abstract class Bike {

private String model;

private int gearCount;

private double wheelSize;

public Bike(String model, int gearCount, double wheelSize) {

this.model = model;

this.gearCount = gearCount;

this.wheelSize = wheelSize;

}

public String getModel() {

return model;

}

public void setModel(String model) {

this.model = model;

}

public int getGearCount() {

return gearCount;

}

public void setGearCount(int gearCount) {

this.gearCount = gearCount;

}

public double getWheelSize() {

return wheelSize;

}

public void setWheelSize(double wheelSize) {

this.wheelSize = wheelSize;

}

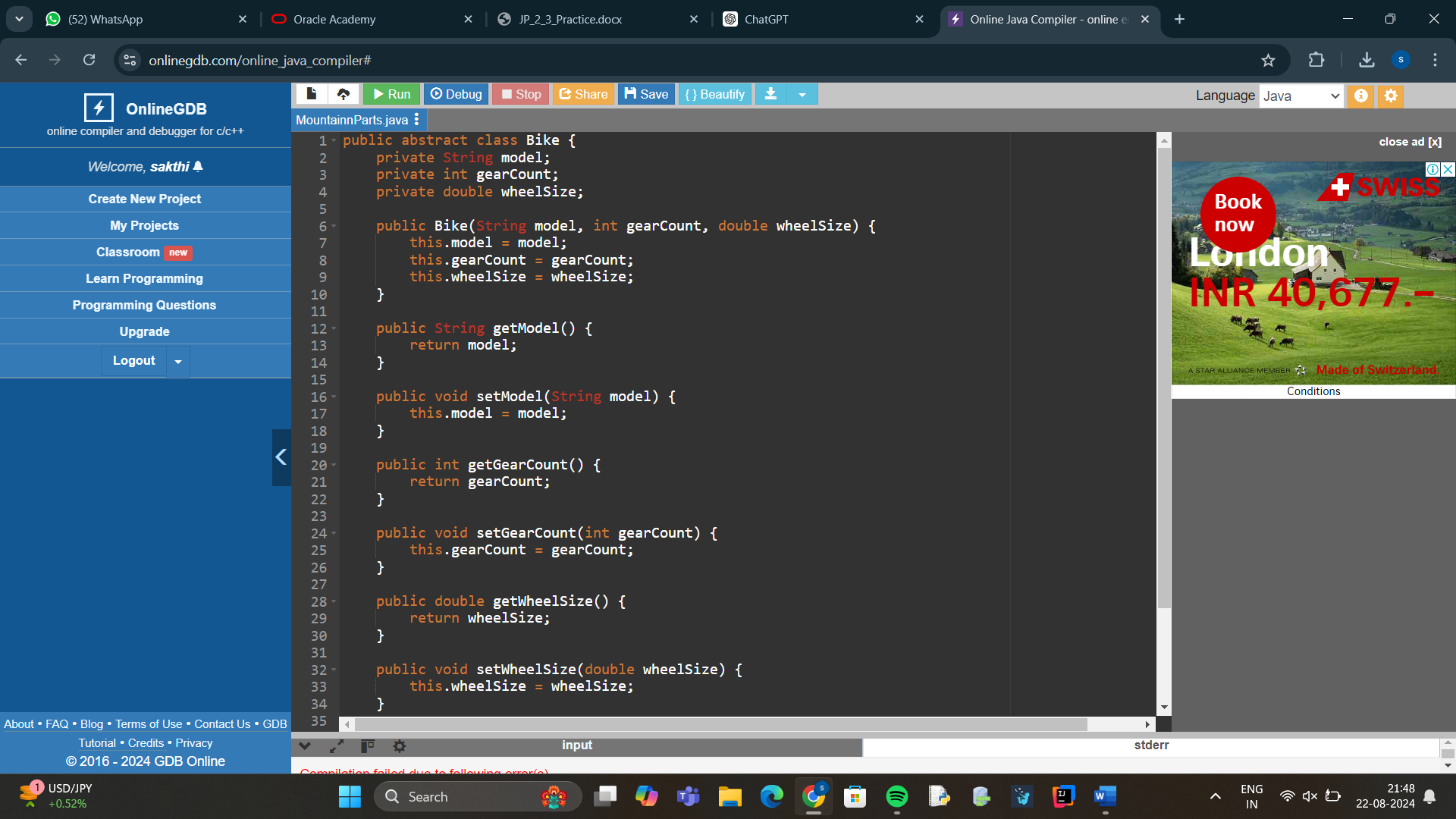
@Override

public String toString() {

return "Bike [Model=" + model + ", Gear Count=" + gearCount + ", Wheel Size=" + wheelSize + "]";

}

}



4.

public class Main {

public static void main(String[] args) {

// Instantiating bikes

Bike bike1 = new MountainBike("Mountain X", 18, 27.5, "Full Suspension");

Bike bike2 = new RoadBike("Roadster", 21, 28.0);

Bike bike3 = new MountainBike("Mountain Y", 20, 29.0, "Hardtail");

Bike bike4 = new RoadBike("Speedster", 22, 30.0); // bike4 code to be removed

// Displaying bike details

System.out.println(bike1);

System.out.println(bike2);

System.out.println(bike3);

System.out.println(bike4); // bike4 code to be removed

}

}

5.

public abstract class Bike {

private String model;

private int gearCount;

private double wheelSize;

public Bike(String model, int gearCount, double wheelSize) {

this.model = model;

this.gearCount = gearCount;

this.wheelSize = wheelSize;

}

@Override

public String toString() {

return "Bike [Model=" + model + ", Gear Count=" + gearCount + ", Wheel Size=" + wheelSize + "]";

}

}

public class MountainBike extends Bike {

private String suspensionType;

public MountainBike(String model, int gearCount, double wheelSize, String suspensionType) {

super(model, gearCount, wheelSize);

this.suspensionType = suspensionType;

}

@Override

public String toString() {

return super.toString() + ", Suspension Type=" + suspensionType + "]";

}

}

public class RoadBike extends Bike {

public RoadBike(String model, int gearCount, double wheelSize) {

super(model, gearCount, wheelSize);

}

@Override

public String toString() {

return super.toString() + "]";

}

}

public class Main {

public static void main(String[] args) {

Bike bike1 = new MountainBike("Mountain X", 18, 27.5, "Full Suspension");

Bike bike2 = new RoadBike("Roadster", 21, 28.0);

Bike bike3 = new MountainBike("Mountain Y", 20, 29.0, "Hardtail");

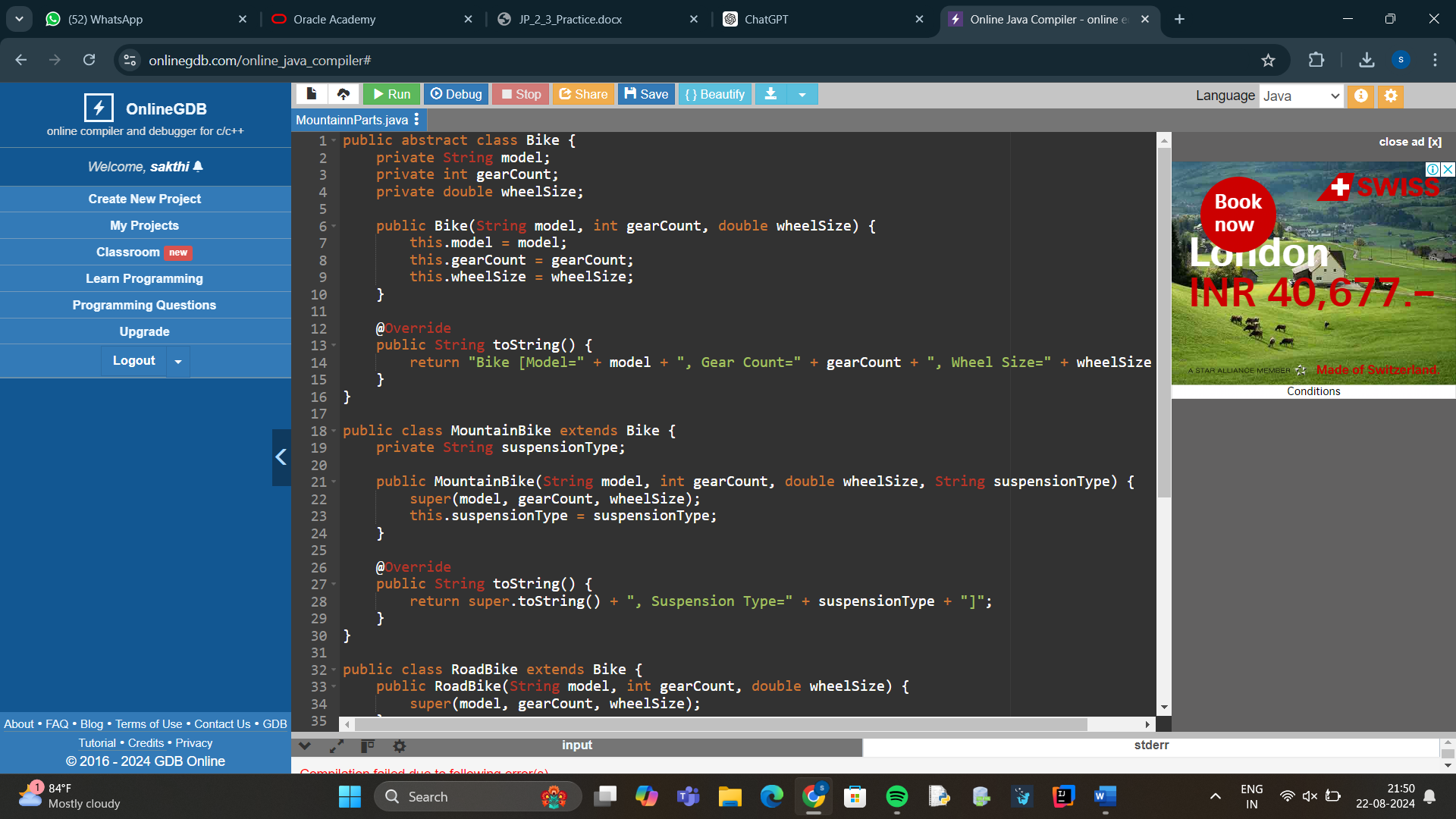
System.out.println(bike1);

System.out.println(bike2);

System.out.println(bike3);

}

}



6.PART A:

public class Animal {

public void makeNoise() {

System.out.println("talk");

}

}

public class Dog extends Animal {

public void makeNoise() {

System.out.println("Bark");

}

}

Explanation:

1. animal.makeNoise();
   * This calls the makeNoise method from the Animal class.
   * Output: talk
2. dog.makeNoise();
   * This calls the makeNoise method from the Dog class, since dog is an instance of Dog.
   * Output: Bark
3. animaldog.makeNoise();
   * animaldog is declared as type Animal, but it refers to an instance of Dog. Due to dynamic method dispatch (runtime polymorphism), the makeNoise method from Dog is called.
   * Output: Bark

PART B:

if (animal instanceof Animal)

System.out.println("animal is Animal");

if (dog instanceof Animal)

System.out.println("dog is Animal");

if (animaldog instanceof Animal)

System.out.println("animaldog is Animal");

if (animal instanceof Dog)

System.out.println("animal is Dog");

Explanation:

1. if (animal instanceof Animal)
   * animal is indeed an instance of Animal.
   * Output: animal is Animal
2. if (dog instanceof Animal)
   * dog is an instance of Dog, which extends Animal, so dog is also an instance of Animal.
   * Output: dog is Animal
3. if (animaldog instanceof Animal)
   * animaldog is declared as Animal, and it refers to an instance of Dog. Therefore, animaldog is an instance of Animal.
   * Output: animaldog is Animal
4. if (animal instanceof Dog)
   * animal is an instance of Animal, not Dog. Therefore, animal is not an instance of Dog.
   * Output: (No output; the condition is false)

Summary

Outputs:

* From Part (a):
  + talk
  + Bark
  + Bark
* From Part (b):
  + animal is Animal
  + dog is Animal
  + animaldog is Animal