# JAVA PRACTICE PROGRAM

**MODULE-1 (7.2)**

1. Create a segment of code that initializes a public class Fish. Let the class contain a String typeOfFish, and an integer friendliness. Do not set values to these variables yet. These are instance variables and will be set inside the class constructors.

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

}

public Fish(String typeOfFish, int friendliness) {

this.typeOfFish = typeOfFish;

this.friendliness = friendliness;

}

public static void main(String[] args) {

Fish goldfish = new Fish("Goldfish", 7);

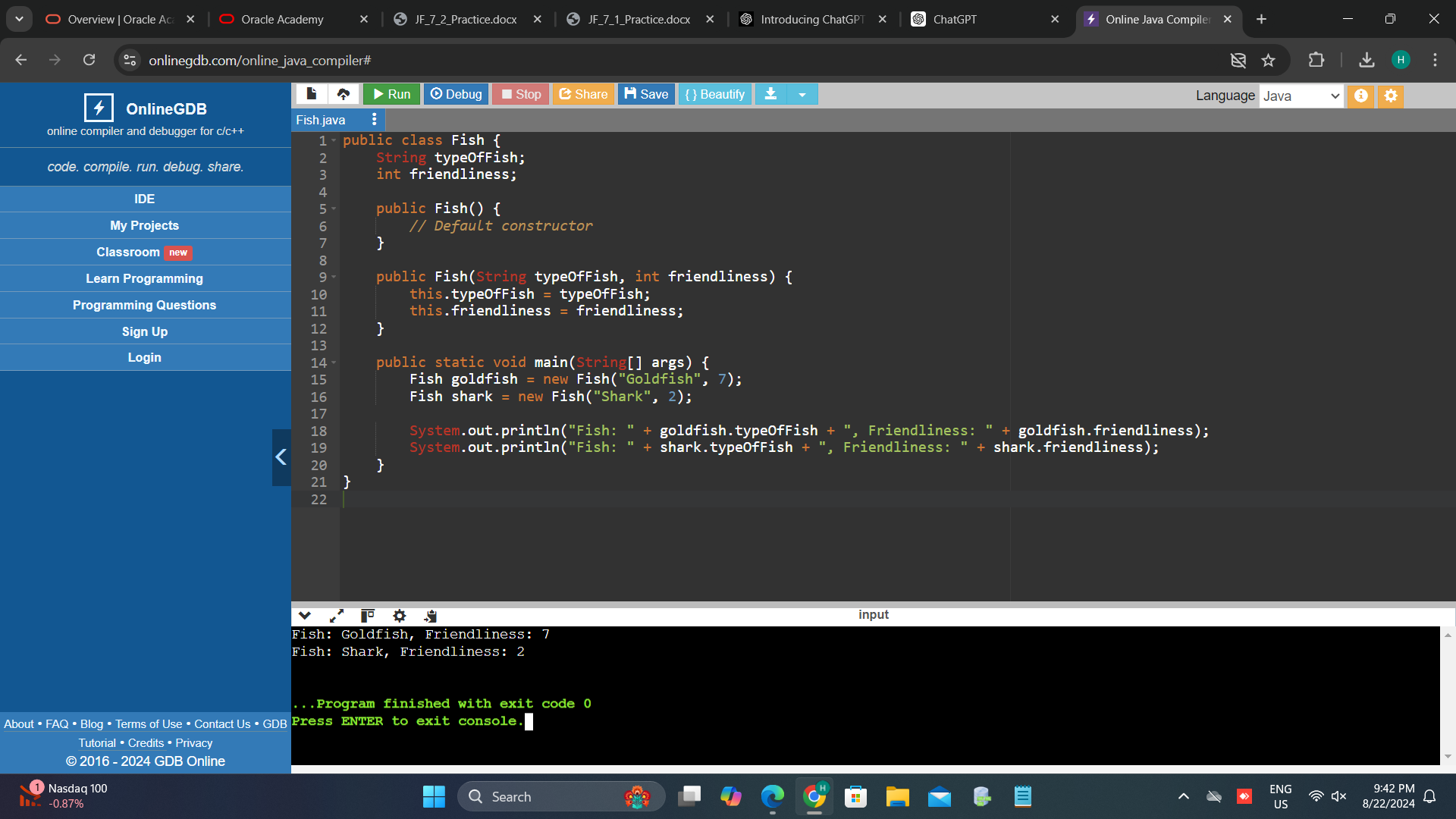
Fish shark = new Fish("Shark", 2);

System.out.println("Fish: " + goldfish.typeOfFish + ", Friendliness: " + goldfish.friendliness);

System.out.println("Fish: " + shark.typeOfFish + ", Friendliness: " + shark.friendliness);

}

}



1. Create a public constructor (a method with the same name as the class) inside the class Fish. This constructor should take in no arguments. Inside the constructor, set typeOfFish to “Unknown” and friendliness to 3, which we are assuming is the generic friendliness of fish.

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String typeOfFish, int friendliness) {

this.typeOfFish = typeOfFish;

this.friendliness = friendliness;

}

public static void main(String[] args) {

Fish unknownFish = new Fish();

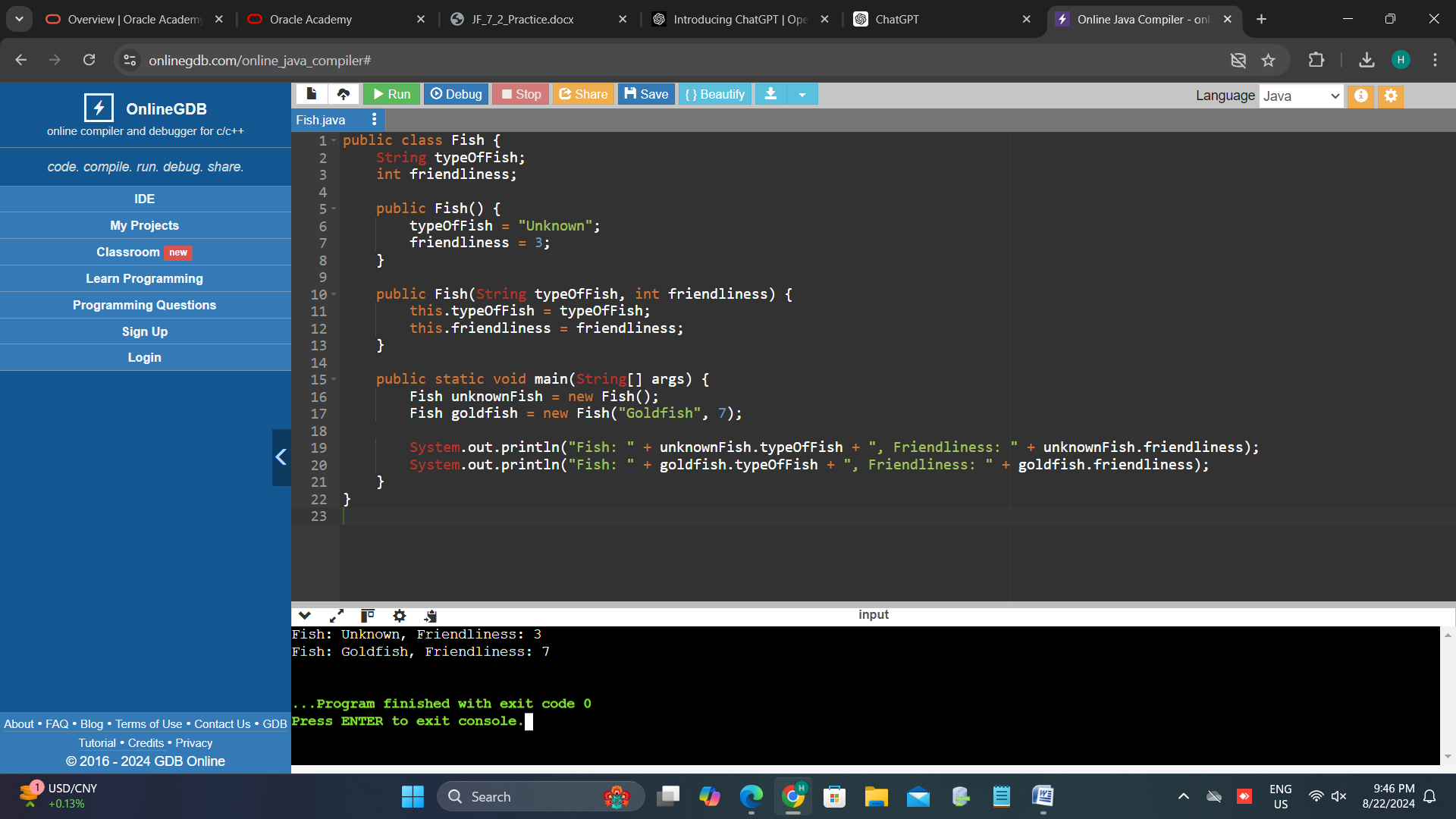
Fish goldfish = new Fish("Goldfish", 7);

System.out.println("Fish: " + unknownFish.typeOfFish + ", Friendliness: " + unknownFish.friendliness);

System.out.println("Fish: " + goldfish.typeOfFish + ", Friendliness: " + goldfish.friendliness);

}

}



1. Create another public constructor inside the class Fish. Have this constructor take in a string t and an integer f. Let typeOfFish equal t, and friendliness equal f.

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public static void main(String[] args) {

Fish unknownFish = new Fish();

Fish goldfish = new Fish("Goldfish", 7);

Fish shark = new Fish("Shark", 2);

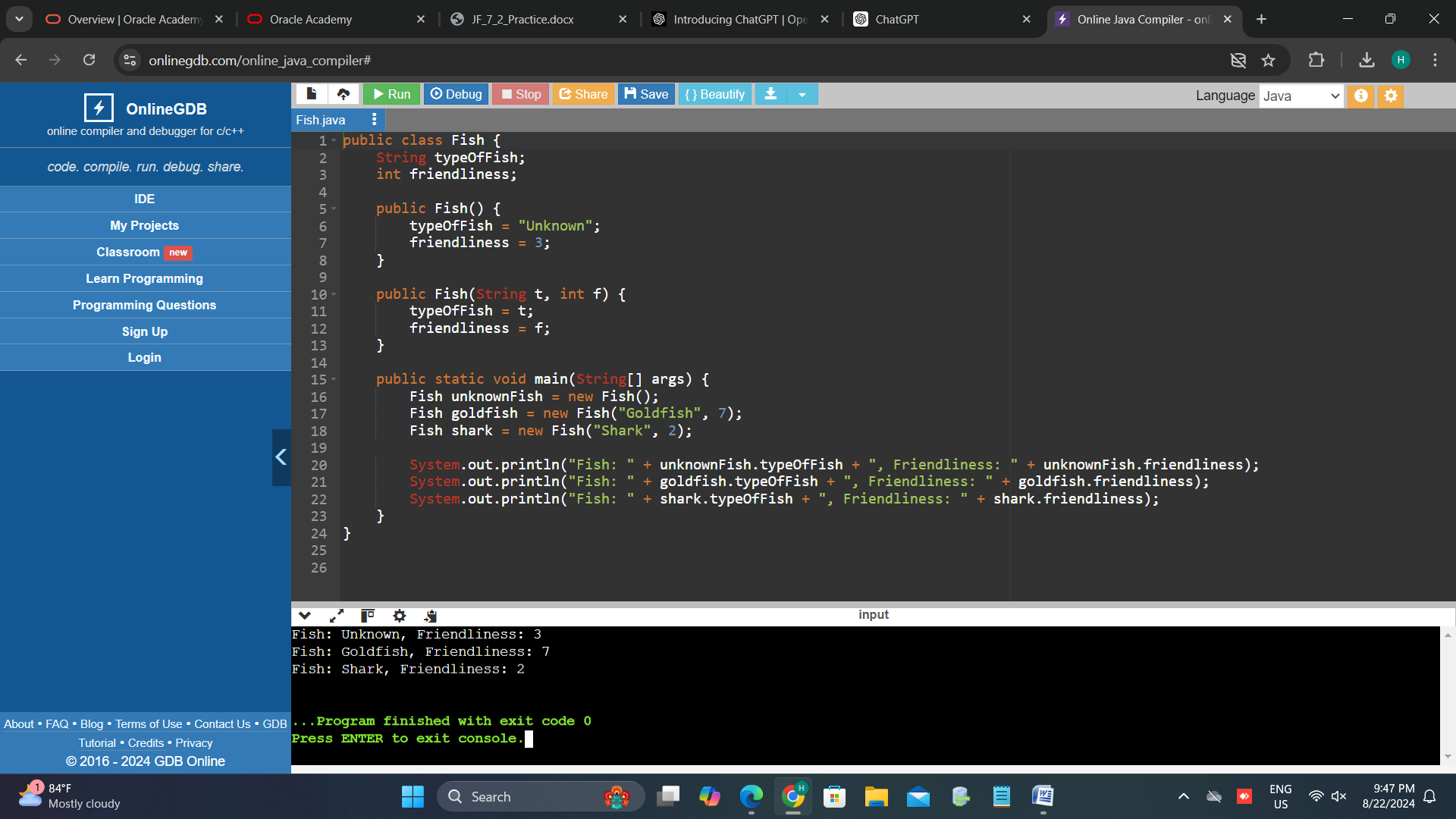
System.out.println("Fish: " + unknownFish.typeOfFish + ", Friendliness: " + unknownFish.friendliness);

System.out.println("Fish: " + goldfish.typeOfFish + ", Friendliness: " + goldfish.friendliness);

System.out.println("Fish: " + shark.typeOfFish + ", Friendliness: " + shark.friendliness);

}

}



1. Explain why is is possible to have more than one constructor with the same name and different arguments.

In Java, it's possible to have more than one constructor with the same name but different arguments because of a concept called **constructor overloading**. This is a form of **polymorphism**, which allows methods (or constructors) in a class to have the same name but differ in the number, types, or order of their parameters.

**How Constructor Overloading Works:**

1. **Same Name, Different Signatures**:
   * Constructors in a class must have the same name as the class, but each constructor can have a different parameter list, known as the "method signature."
   * Java distinguishes between these constructors based on the number and types of arguments you pass when you create an object. This is how it knows which constructor to call.
2. **Flexibility in Object Creation**:
   * Constructor overloading provides flexibility to create objects in different ways. For example, you might want to create an object with default values, or you might want to specify some or all of the attributes during object creation.
   * In the **Fish** example, one constructor allows creating a **Fish** object with default values (**Fish()**), while the other allows specifying the **typeOfFish** and **friendliness** directly (**Fish(String t, int f)**).

**Benefits:**

* **Code Clarity and Convenience**: Users of the class can create objects with varying levels of detail based on their needs, without needing to know how the class internally manages these different constructors.
* **Maintainability**: It keeps the class clean and avoids the need for different named methods or complex object initialization logic.

1. Create a method inside the class Fish called getFriendliness(), which takes in no arguments and returns the friendliness level of the fish

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public int getFriendliness() {

return friendliness;

}

public static void main(String[] args) {

Fish unknownFish = new Fish();

Fish goldfish = new Fish("Goldfish", 7);

Fish shark = new Fish("Shark", 2);

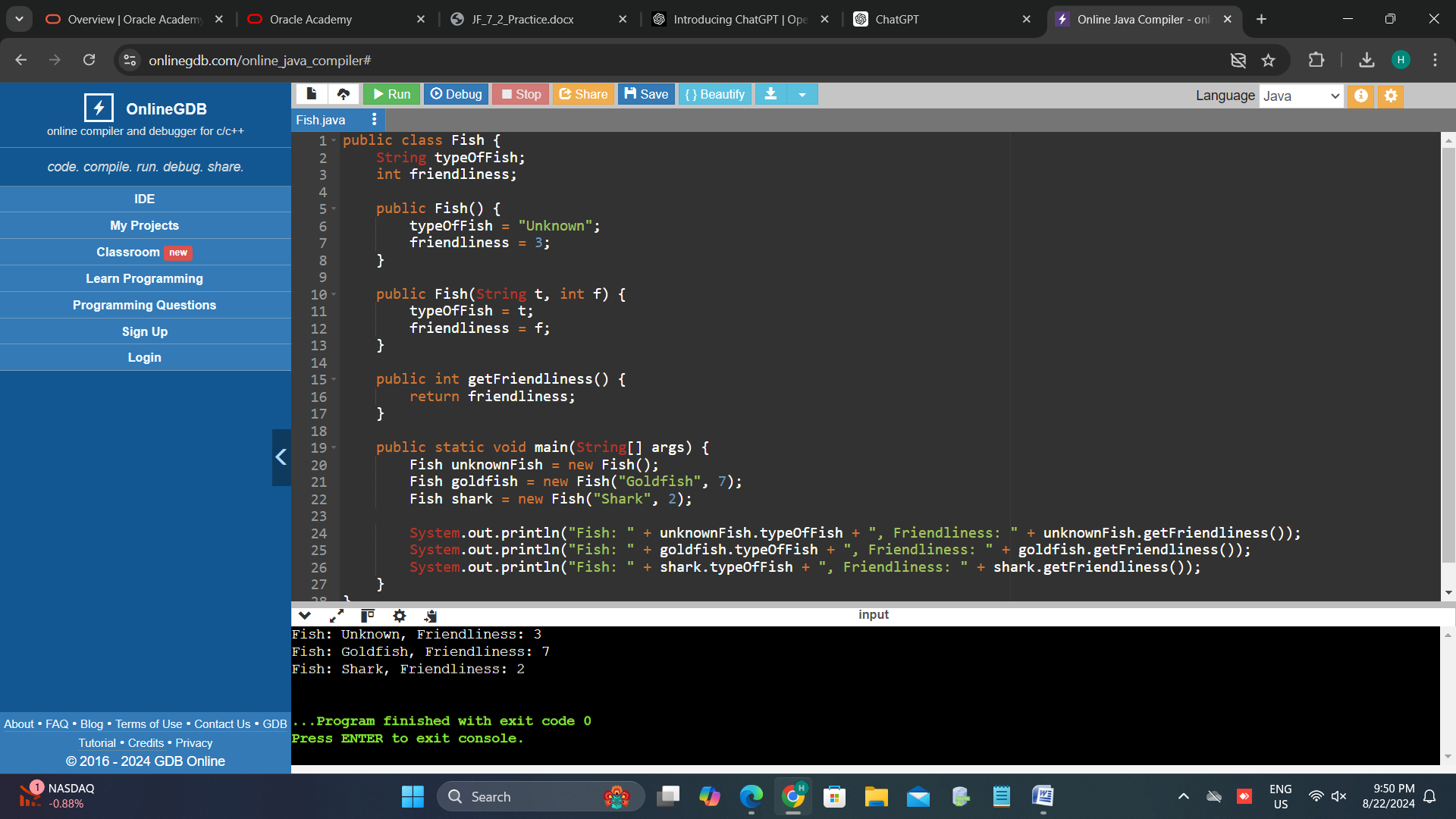
System.out.println("Fish: " + unknownFish.typeOfFish + ", Friendliness: " + unknownFish.getFriendliness());

System.out.println("Fish: " + goldfish.typeOfFish + ", Friendliness: " + goldfish.getFriendliness());

System.out.println("Fish: " + shark.typeOfFish + ", Friendliness: " + shark.getFriendliness());

}

**}**

****

1. Write a segment of code that initializes 2 new fish as defined below: a. Fish 1: Name – Amber, Type – AngelFish, Friendliness level – 5 (very friendly) b. Fish 2: Name – James, Type – Guppy, Friendliness level – 3 (neutral)

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public int getFriendliness() {

return friendliness;

}

public static void main(String[] args) {

Fish fish1 = new Fish("AngelFish", 5); // Amber

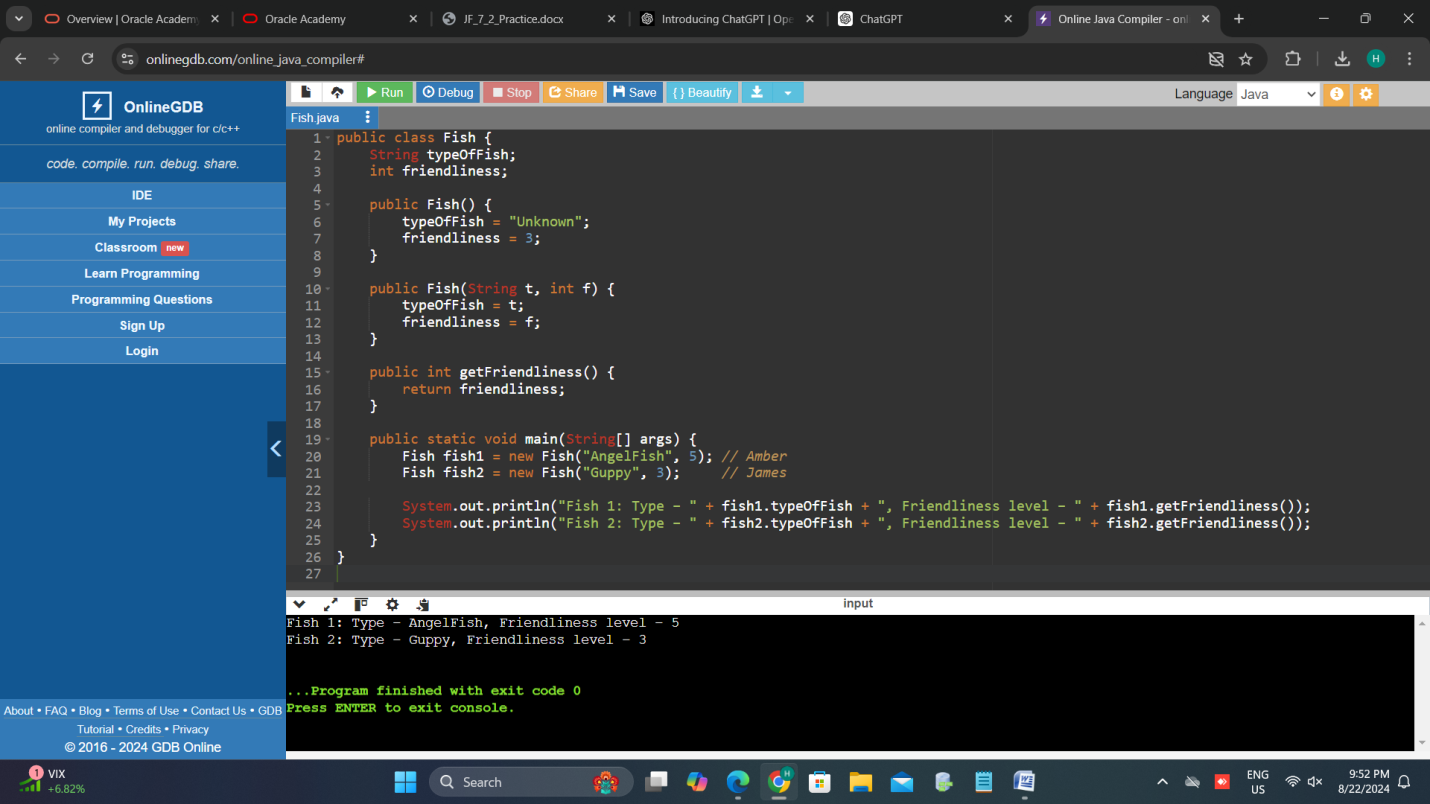
Fish fish2 = new Fish("Guppy", 3); // James

System.out.println("Fish 1: Type – " + fish1.typeOfFish + ", Friendliness level – " + fish1.getFriendliness());

System.out.println("Fish 2: Type – " + fish2.typeOfFish + ", Friendliness level – " + fish2.getFriendliness());

}

}

****

1. Create a method nicestFish that takes in two fish as parameters, compares the friendliness level of two fish, and returns the fish with the higher friendliness. Test this method with the fish defined in problem 6. (Friendliness scale: 1 mean, 2 not friendly, 3 neutral, 4 friendly, 5 very friendly) Hint: fishName.getFriendliness() gives you the integer number of the friendliness of fishName. You have already created getFriendliness() in problem 5.

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public int getFriendliness() {

return friendliness;

}

public static Fish nicestFish(Fish fish1, Fish fish2) {

if (fish1.getFriendliness() > fish2.getFriendliness()) {

return fish1;

} else {

return fish2;

}

}

public static void main(String[] args) {

Fish fish1 = new Fish("AngelFish", 5); // Amber

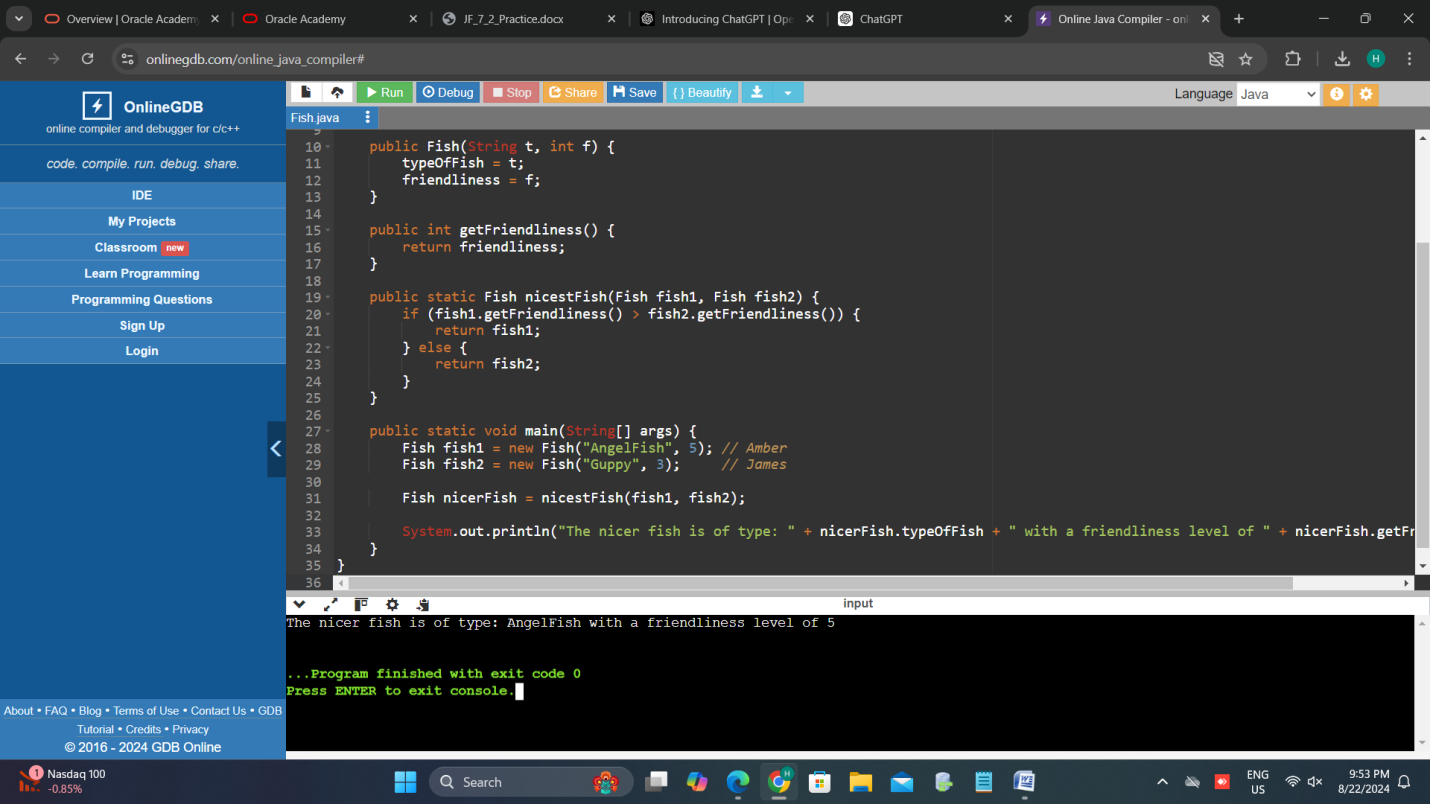
Fish fish2 = new Fish("Guppy", 3); // James

Fish nicerFish = nicestFish(fish1, fish2);

System.out.println("The nicer fish is of type: " + nicerFish.typeOfFish + " with a friendliness level of " + nicerFish.getFriendliness());

}

}



1. Modify the method nicestFish() to take be a variable argument method that takes in a variable number of fish and returns the nicest fish out of the fish it is given. Hint: Inside of the method, create a new fish called temp. Set temp equal to the first fish passed into the method. Use a for loop to go through all the fish passed into the method and if you discover a fish that is more friendly than temp, set temp equal to that fish. After the for loop is complete, temp should be the friendliest fish. Return temp.

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public int getFriendliness() {

return friendliness;

}

public static Fish nicestFish(Fish... fishArray) {

if (fishArray.length == 0) {

return null; // or throw an exception if no fish are provided

}

Fish temp = fishArray[0];

for (Fish fish : fishArray) {

if (fish.getFriendliness() > temp.getFriendliness()) {

temp = fish;

}

}

return temp;

}

public static void main(String[] args) {

Fish fish1 = new Fish("AngelFish", 5); // Amber

Fish fish2 = new Fish("Guppy", 3); // James

Fish fish3 = new Fish("Goldfish", 4); // Goldie

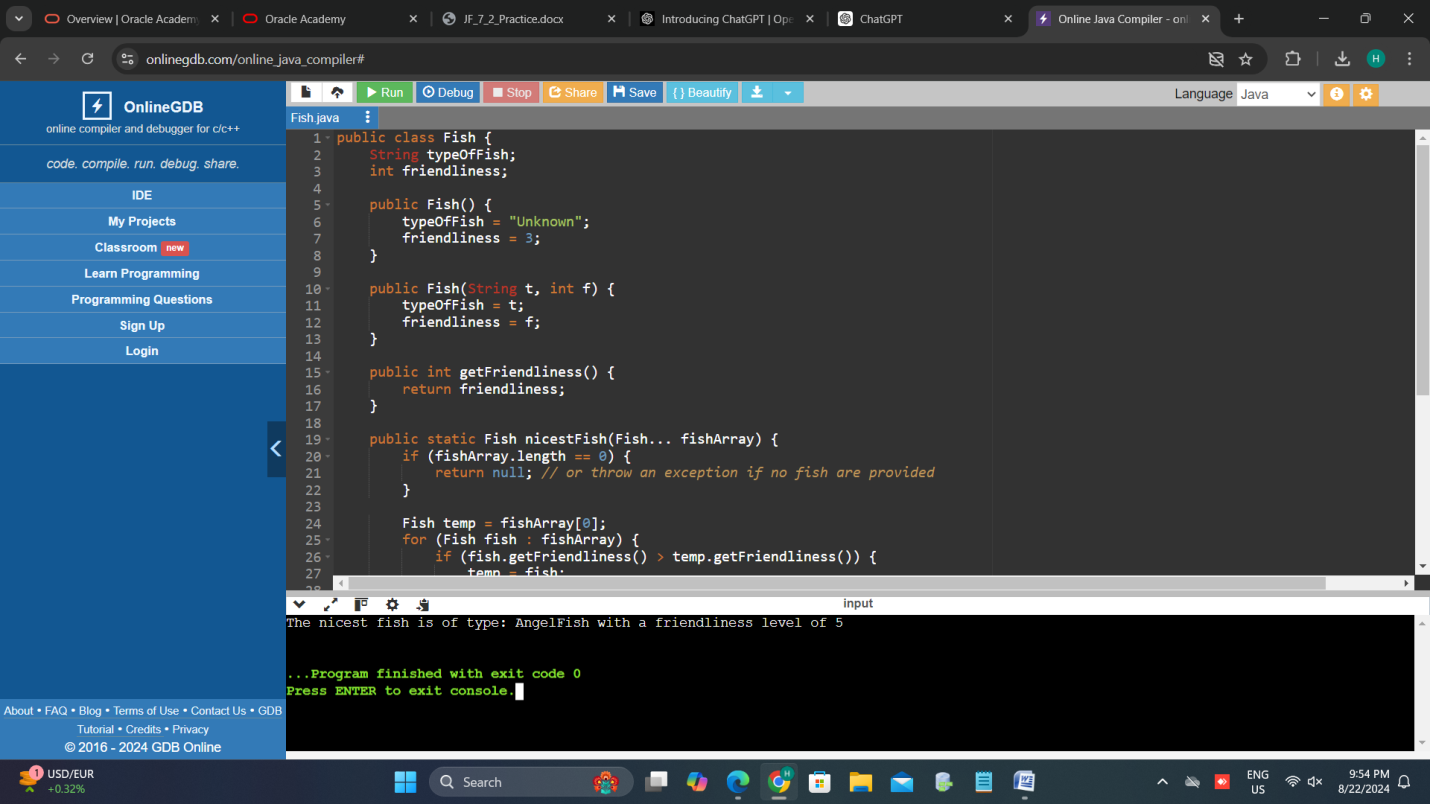
Fish fish4 = new Fish("Betta", 2); // Betta

Fish nicest = nicestFish(fish1, fish2, fish3, fish4);

System.out.println("The nicest fish is of type: " + nicest.typeOfFish + " with a friendliness level of " + nicest.getFriendliness());

}

}



1. Test your method nicestFish() with the fish described in problem 6. Which fish is returned?

**CODE:**

public class Fish {

String typeOfFish;

int friendliness;

public Fish() {

typeOfFish = "Unknown";

friendliness = 3;

}

public Fish(String t, int f) {

typeOfFish = t;

friendliness = f;

}

public int getFriendliness() {

return friendliness;

}

public static Fish nicestFish(Fish... fishArray) {

if (fishArray.length == 0) {

return null;

}

Fish temp = fishArray[0];

for (Fish fish : fishArray) {

if (fish.getFriendliness() > temp.getFriendliness()) {

temp = fish;

}

}

return temp;

}

public static void main(String[] args) {

Fish fish1 = new Fish("AngelFish", 5);

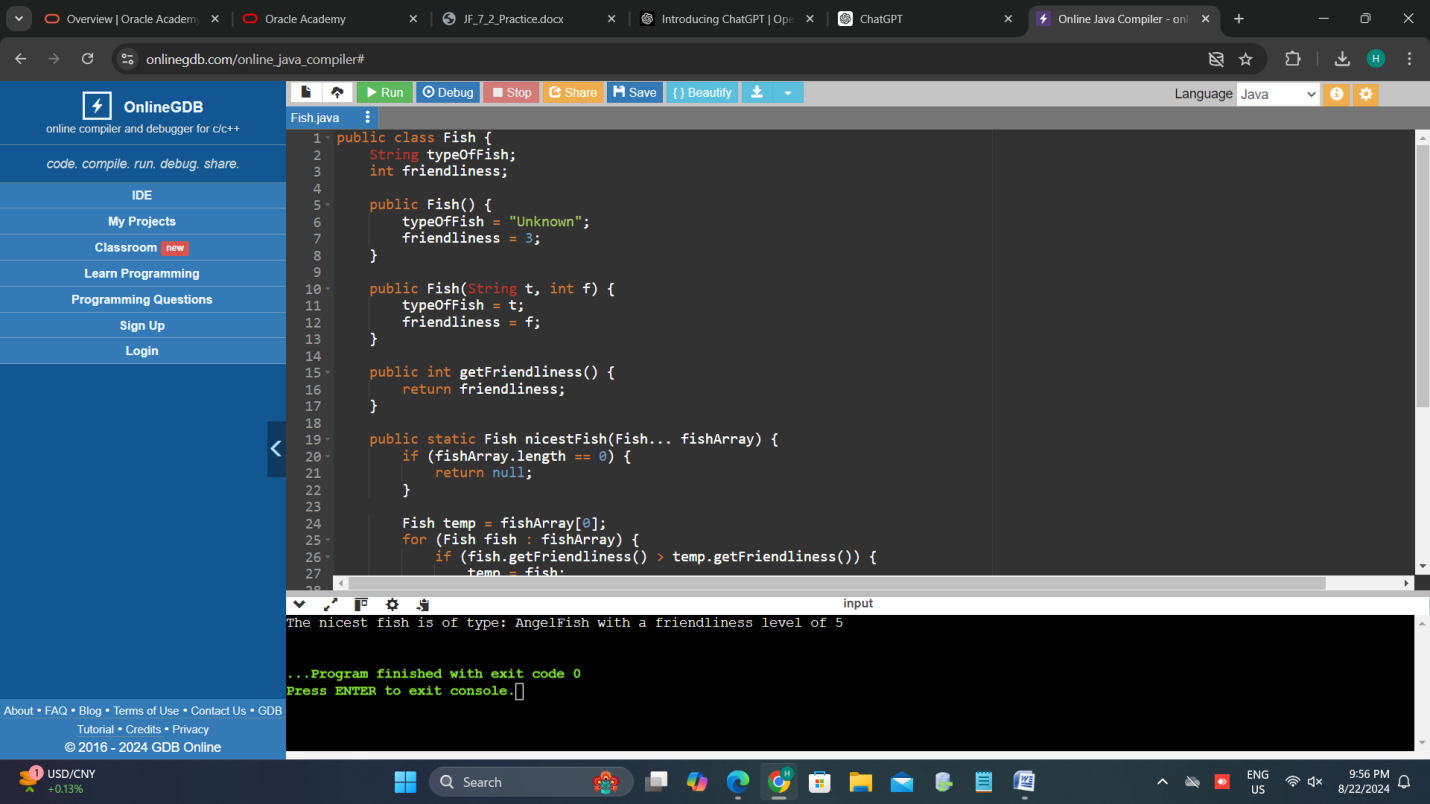
Fish fish2 = new Fish("Guppy", 3);

Fish nicest = nicestFish(fish1, fish2);

System.out.println("The nicest fish is of type: " + nicest.typeOfFish + " with a friendliness level of " + nicest.getFriendliness());

}

}



1. Determine the best access modifier for each of the following situations: a. A class Employee records the name, address, salary, and phone number. b. An adding method inside of a class BasicMath that is also used in the Algebra class.

a. Class `Employee` records:

Access Modifier: `private` (for instance variables)

Explanation:

-Private access modifier is the best choice for the instance variables `name`, `address`, `salary`, and `phone number` because it restricts direct access to these variables from outside the class. This promotes encapsulation by ensuring that these fields can only be modified or accessed through public getter and setter methods, which can provide validation and control over how the data is managed.

- Public getter and setter methods should be provided to allow controlled access to these private fields.

b. An `adding` method inside of a class `BasicMath` that is also used in the `Algebra` class:

Access Modifier: `public` or `protected`

Explanation:

- Public access modifier is suitable if the `adding` method needs to be accessible from any other class, including the `Algebra` class and potentially other classes or packages.

- Protected access modifier can be used if the `adding` method should be accessible to subclasses of `BasicMath` (like `Algebra`) and classes within the same package, but not necessarily to all classes. This provides more restricted access compared to `public`, which might be appropriate if the method is only intended to be used within a certain hierarchy or package.