**Exercise 1:**

**Create an interface Drawable with a method draw(). Implement this interface in classes Circle and Square. Explain the concept of interfaces and how they are used in this example**.

Answer 1:

interface Drawable {

void draw();

}

class Circle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a circle.");

}

}

class Square implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a square.");

}

}

Explanation:

An interface in Java is a way to define a contract that a class must adhere to. It specifies a set of method signatures that classes implementing the interface must provide concrete implementations for. In this example, the Drawable interface defines a single method draw(). The classes Circle and Square implement the Drawable interface, providing their own specific implementations for the draw() method.

**Exercise 2:**

**Create an interface Playable with a method playSound(). Implement this interface in classes Piano and Guitar. Explain the benefits of using interfaces for implementing common behavior.**

Answer 2:

interface Playable {

void playSound();

}

class Piano implements Playable {

@Override

public void playSound() {

System.out.println("Piano sound is played.");

}

}

class Guitar implements Playable {

@Override

public void playSound() {

System.out.println("Guitar sound is played.");

}

}

Explanation:

Using interfaces for implementing common behavior allows classes to share a common contract without worrying about the specifics of each class. In this case, both Piano and Guitar implement the Playable interface, ensuring that they provide their own implementations for the playSound() method. This helps achieve code consistency and maintainability.

**Exercise 3:**

**Create an interface Shape with methods calculateArea() and calculatePerimeter(). Implement this interface in classes Circle and Rectangle. Explain how interfaces allow multiple inheritance**.

Answer 3:

interface Shape {

double calculateArea();

double calculatePerimeter();

}

class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

@Override

public double calculatePerimeter() {

return 2 \* Math.PI \* radius;

}

}

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

public double calculateArea() {

return length \* width;

}

@Override

public double calculatePerimeter() {

return 2 \* (length + width);

}

}

Explanation:

Interfaces in Java allow multiple inheritance, meaning a class can implement multiple interfaces. In this case, the Circle and Rectangle classes both implement the Shape interface, which defines two methods: calculateArea() and calculatePerimeter(). By implementing this interface, each class provides its own calculations for these methods, showcasing the flexibility of interfaces for achieving multiple inheritance-like behavior.

**Exercise 4:**

**Create an interface Movable with methods moveLeft(), moveRight(), moveUp(), and moveDown(). Implement this interface in a class Player. Explain how interfaces help in achieving a consistent API for different classes.**

Answer 4:

interface Movable {

void moveLeft();

void moveRight();

void moveUp();

void moveDown();

}

class Player implements Movable {

private int x;

private int y;

// Constructor and other methods

@Override

public void moveLeft() {

x--;

}

@Override

public void moveRight() {

x++;

}

@Override

public void moveUp() {

y++;

}

@Override

public void moveDown() {

y--;

}

}

Explanation:

Interfaces provide a consistent API (method signatures) that classes can adhere to, ensuring that specific behaviors are implemented uniformly across different classes. In this case, the Movable interface defines methods for movement in different directions. The Player class implements this interface, providing its own implementations for each movement method. This way, regardless of the actual class, the API for movement remains consistent.

**Exercise 5:**

**Create an interface Logger with a method log(String message). Implement this interface in classes ConsoleLogger and FileLogger. Demonstrate how you can achieve polymorphism using interfaces.**

Answer 5:

interface Logger {

void log(String message);

}

class ConsoleLogger implements Logger {

@Override

public void log(String message) {

System.out.println("Console Log: " + message);

}

}

class FileLogger implements Logger {

@Override

public void log(String message) {

// Code to log message to a file

System.out.println("File Log: " + message);

}

}

Explanation:

Polymorphism refers to the ability of different classes to be treated as instances of a common parent class or interface. In this case, the Logger interface serves as the common contract for logging. Both ConsoleLogger and FileLogger implement the Logger interface, which means you can use instances of either class interchangeably when you want to log messages, demonstrating the concept of polymorphism.

**Exercise 6:**

**Create an interface Database with methods connect() and disconnect(). Implement this interface in classes MySQLDatabase and OracleDatabase. Illustrate how interfaces can be used to switch between different implementations seamlessly**.

Answer 6:

interface Database {

void connect();

void disconnect();

}

class MySQLDatabase implements Database {

@Override

public void connect() {

System.out.println("Connected to MySQL database.");

}

@Override

public void disconnect() {

System.out.println("Disconnected from MySQL database.");

}

}

class OracleDatabase implements Database {

@Override

public void connect() {

System.out.println("Connected to Oracle database.");

}

@Override

public void disconnect() {

System.out.println("Disconnected from Oracle database.");

}

}

Explanation:

Interfaces provide a way to define a common contract that multiple classes can adhere to. In this scenario, both MySQLDatabase and OracleDatabase classes implement the Database interface. By adhering to the same interface, you can easily switch between different database implementations without changing the rest of your code, showcasing the concept of abstraction and modularity.

**Exercise 7:**

**Create an interface Resizable with a method resize(double factor). Implement this interface in a class ResizableCircle. Explain how interfaces can introduce new behavior to existing classes.**

Answer 7:

interface Resizable {

void resize(double factor);

}

class ResizableCircle implements Resizable {

private double radius;

public ResizableCircle(double radius) {

this.radius = radius;

}

@Override

public void resize(double factor) {

radius \*= factor;

}

// Other methods and properties for the circle

}

Explanation:

Interfaces can be used to introduce new behavior to existing classes without modifying their structure. In this case, the Resizable interface adds the ability to resize a shape. The ResizableCircle class implements the Resizable interface and provides a resize() method that modifies the radius based on the resizing factor. This showcases how interfaces can enhance the capabilities of classes without directly modifying their code.

**Exercise 8:**

**Create an interface ElectricDevice with a method turnOn() and turnOff(). Implement this interface in classes Fan and Light. Explain how interfaces allow you to define a contract for unrelated classes.**

Answer 8:

interface ElectricDevice {

void turnOn();

void turnOff();

}

class Fan implements ElectricDevice {

@Override

public void turnOn() {

System.out.println("Fan turned on.");

}

@Override

public void turnOff() {

System.out.println("Fan turned off.");

}

}

class Light implements ElectricDevice {

@Override

public void turnOn() {

System.out.println("Light turned on.");

}

@Override

public void turnOff() {

System.out.println("Light turned off.");

}

}

Explanation:

Interfaces allow you to define a contract that unrelated classes can adhere to. In this example, the ElectricDevice interface provides a common structure for classes that represent electrical devices, even if those devices have entirely different functionalities. Both Fan and Light classes implement this interface and provide their own implementations for the turnOn() and turnOff() methods, demonstrating how interfaces enable consistency among unrelated classes.

**Exercise 9:**

**Create an interface PaymentProcessor with methods processPayment(double amount) and refundPayment(double amount). Implement this interface in classes CreditCardProcessor and PayPalProcessor. Explain how interfaces promote loose coupling.**

Answer 9:

interface PaymentProcessor {

void processPayment(double amount);

void refundPayment(double amount);

}

class CreditCardProcessor implements PaymentProcessor {

@Override

public void processPayment(double amount) {

// Process payment using credit card

}

@Override

public void refundPayment(double amount) {

// Refund payment using credit card

}

}

class PayPalProcessor implements PaymentProcessor {

@Override

public void processPayment(double amount) {

// Process payment using PayPal

}

@Override

public void refundPayment(double amount) {

// Refund payment using PayPal

}

}

Explanation:

Loose coupling is a design principle that promotes independence between components. Interfaces play a significant role in achieving loose coupling, as seen in this example. The PaymentProcessor interface defines a contract for processing and refunding payments. Both CreditCardProcessor and PayPalProcessor implement this interface, ensuring that they provide the necessary payment-related functionality independently. This loose coupling allows for easy replacement of one processor with another without affecting the rest of the system.

**Exercise 10:**

**Create an interface Animal with a method makeSound(). Implement this interface in classes Dog and Cat. Explain how interfaces facilitate code organization and polymorphism.**

Answer 10:

interface Animal {

void makeSound();

}

class Dog implements Animal {

@Override

public void makeSound() {

System.out.println("Dog barks.");

}

}

class Cat implements Animal {

@Override

public void makeSound() {

System.out.println("Cat meows.");

}

}

Explanation:

Interfaces provide a way to group related behavior together, which improves code organization. In this case, the Animal interface groups different animals under a common contract of making a sound. The Dog and Cat classes implement this interface, providing their own sound implementations. Interfaces also enable polymorphism, allowing you to treat instances of Dog and Cat as instances of the Animal interface, providing a consistent API for various animal types.

**Exercise 11:**

**Create an interface Employee with methods calculateSalary() and displayInfo(). Implement this interface in classes Manager and Developer. Explain how interfaces enforce certain behavior across different classes.**

Answer 11:

interface Employee {

double calculateSalary();

void displayInfo();

}

class Manager implements Employee {

@Override

public double calculateSalary() {

// Calculation logic for manager's salary

return 50000.0;

}

@Override

public void displayInfo() {

System.out.println("Manager's information.");

}

}

class Developer implements Employee {

@Override

public double calculateSalary() {

// Calculation logic for developer's salary

return 60000.0;

}

@Override

public void displayInfo() {

System.out.println("Developer's information.");

}

}

Explanation:

Interfaces provide a contract that classes must adhere to, ensuring that specific behaviors are implemented across different classes. In this scenario, the Employee interface defines methods for calculating salary and displaying information. Both Manager and Developer classes implement this interface, guaranteeing that they provide implementations for both methods. This helps enforce a certain structure and behavior while allowing customization based on the specific class.

**Exercise 12:**

**Create an interface Playable with methods play(), pause(), and stop(). Implement this interface in classes AudioPlayer and VideoPlayer. Illustrate how interfaces can be used to create a unified interface for different types of media players.**

Answer 12:

interface Playable {

void play();

void pause();

void stop();

}

class AudioPlayer implements Playable {

@Override

public void play() {

System.out.println("Audio is playing.");

}

@Override

public void pause() {

System.out.println("Audio playback paused.");

}

@Override

public void stop() {

System.out.println("Audio playback stopped.");

}

}

class VideoPlayer implements Playable {

@Override

public void play() {

System.out.println("Video is playing.");

}

@Override

public void pause() {

System.out.println("Video playback paused.");

}

@Override

public void stop() {

System.out.println("Video playback stopped.");

}

}

Explanation:

Interfaces allow you to define a common set of methods that different classes can implement, even if those classes have varying functionalities. The Playable interface defines methods related to media playback. Both AudioPlayer and VideoPlayer classes implement this interface, providing their own implementations for the playback-related methods. This approach creates a unified interface for different types of media players, promoting code consistency.

**Exercise 13:**

**Create an interface BankAccount with methods deposit(double amount) and withdraw(double amount). Implement this interface in classes SavingsAccount and CheckingAccount. Explain how interfaces help in achieving code reusability and polymorphism.**

Answer 13:

interface BankAccount {

void deposit(double amount);

void withdraw(double amount);

}

class SavingsAccount implements BankAccount {

private double balance;

@Override

public void deposit(double amount) {

balance += amount;

}

@Override

public void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

} else {

System.out.println("Insufficient funds.");

}

}

}

class CheckingAccount implements BankAccount {

private double balance;

@Override

public void deposit(double amount) {

balance += amount;

}

@Override

public void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

} else {

System.out.println("Insufficient funds.");

}

}

}

Explanation:

Interfaces promote code reusability by providing a common contract that different classes can adhere to. In this case, the BankAccount interface defines methods for depositing and withdrawing money. Both SavingsAccount and CheckingAccount classes implement this interface, reusing the same methods to handle financial transactions. Additionally, interfaces enable polymorphism, allowing you to treat instances of different account types uniformly through the BankAccount interface.

**Exercise 14:**

**Create an interface Comparable with a method compareTo(Object other). Implement this interface in a class Person. Demonstrate how you can use the compareTo() method to compare instances of Person.**

Answer 14:

interface Comparable {

int compareTo(Object other);

}

class Person implements Comparable {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int compareTo(Object other) {

if (!(other instanceof Person)) {

throw new IllegalArgumentException("Object is not a Person.");

}

Person otherPerson = (Person) other;

return this.age - otherPerson.age;

}

}

Explanation:

Interfaces can define a contract for certain behaviors, and in this case, the Comparable interface defines the compareTo() method for comparing objects. The Person class implements this interface, providing its own comparison logic. By using the compareTo() method, you can compare instances of Person based on their ages. This approach allows for easy sorting and comparison of objects in a uniform manner.

**Exercise 15:**

**Create an interface Resizable with methods resizeWidth(double factor) and resizeHeight(double factor). Implement this interface in a class ResizableRectangle. Illustrate how interfaces can help in designing flexible and extensible classes.**

Answer 15:

interface Resizable {

void resizeWidth(double factor);

void resizeHeight(double factor);

}

class ResizableRectangle implements Resizable {

private double width;

private double height;

public ResizableRectangle(double width, double height) {

this.width = width;

this.height = height;

}

@Override

public void resizeWidth(double factor) {

width \*= factor;

}

@Override

public void resizeHeight(double factor) {

height \*= factor;

}

// Other methods and properties for the rectangle

}

Explanation:

Interfaces can provide a blueprint for designing flexible and extensible classes. In this example, the Resizable interface defines methods for resizing width and height. The ResizableRectangle class implements this interface, allowing instances of the class to be resized proportionally. By following this approach, you create classes that adhere to a standard contract while still providing the flexibility to customize behavior, demonstrating the power of interfaces in designing adaptable systems.

**Exercise 16:**

**Create an interface Taxable with a method calculateTax(double income). Implement this interface in classes Individual and Business. Explain how interfaces allow you to provide a common calculation method for different entities.**

Answer 16:

interface Taxable {

double calculateTax(double income);

}

class Individual implements Taxable {

@Override

public double calculateTax(double income) {

// Calculation logic for individual tax

return income \* 0.15;

}

}

class Business implements Taxable {

@Override

public double calculateTax(double income) {

// Calculation logic for business tax

return income \* 0.25;

}

}

Explanation:

Interfaces provide a consistent contract for different classes, even if those classes perform different roles. In this scenario, the Taxable interface defines a method for calculating taxes based on income. Both Individual and Business classes implement this interface, providing their own implementations of the tax calculation logic. This approach allows you to apply tax calculations uniformly across various entities, promoting code consistency and maintainability.

**Exercise 17:**

**Create an interface Shape with methods calculateArea() and calculatePerimeter(). Implement this interface in classes Circle and Rectangle. Demonstrate how interfaces can be used to enforce consistent behavior across a hierarchy of classes.**

Answer 17:

interface Shape {

double calculateArea();

double calculatePerimeter();

}

class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

@Override

public double calculatePerimeter() {

return 2 \* Math.PI \* radius;

}

}

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

public double calculateArea() {

return length \* width;

}

@Override

public double calculatePerimeter() {

return 2 \* (length + width);

}

}

Explanation:

Interfaces can define a common contract that classes must adhere to, ensuring consistent behavior even within a class hierarchy. In this case, the Shape interface defines methods for calculating the area and perimeter of shapes. Both Circle and Rectangle classes implement this interface and provide their own calculations for the area and perimeter. This promotes code consistency and maintainability throughout the hierarchy of shape classes.

**Exercise 18:**

**Create an interface Cooking with a method prepareFood(). Implement this interface in classes Chef and HomeCook. Explain how interfaces allow different classes to provide unique implementations while adhering to a common contract.**

Answer 18:

interface Cooking {

void prepareFood();

}

class Chef implements Cooking {

@Override

public void prepareFood() {

System.out.println("Chef is preparing a gourmet meal.");

}

}

class HomeCook implements Cooking {

@Override

public void prepareFood() {

System.out.println("Home cook is preparing a simple meal.");

}

}

Explanation:

Interfaces provide a common contract while allowing different classes to implement their own unique behaviors. In this example, the Cooking interface defines the prepareFood() method. Both Chef and HomeCook classes implement this interface, but they provide distinct implementations for how food is prepared. This flexibility ensures that each class can customize its behavior while still adhering to the expected contract.

**Exercise 19:**

**Create an interface Displayable with methods show(). Implement this interface in classes Image and Text. Explain how interfaces can be used to handle different types of data in a unified way.**

Answer 19:

interface Displayable {

void show();

}

class Image implements Displayable {

@Override

public void show() {

System.out.println("Displaying an image.");

}

}

class Text implements Displayable {

@Override

public void show() {

System.out.println("Displaying text.");

}

}

Explanation:

Interfaces can be used to handle different types of data uniformly, even if those types have different behavior. The Displayable interface defines a method for showing content. Both Image and Text classes implement this interface, allowing them to be treated as instances of the same interface. This approach simplifies code that needs to work with various types of displayable content in a unified manner.

**Exercise 20:**

**Create an interface Authentication with methods login(String username, String password) and logout(). Implement this interface in classes User and Admin. Explain how interfaces can enforce security-related behavior across different user roles.**

Answer 20:

interface Authentication {

void login(String username, String password);

void logout();

}

class User implements Authentication {

@Override

public void login(String username, String password) {

// User login logic

}

@Override

public void logout() {

// User logout logic

}

}

class Admin implements Authentication {

@Override

public void login(String username, String password) {

// Admin login logic

}

@Override

public void logout() {

// Admin logout logic

}

}

Explanation:

Interfaces can be used to enforce security-related behavior across different user roles, ensuring consistent authentication mechanisms. In this case, the Authentication interface defines methods for login and logout actions. Both User and Admin classes implement this interface, providing their own login and logout logic based on their respective roles. By adhering to the interface, you enforce a common security contract for different user types.