Object-Oriented Programming - Solutions

Problem 1: Create a `Book` Class

```python
# Solution for Problem 1
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def description(self):
        print(f'"{self.title}" by {self.author} has {self.pages} pages.')

# Creating book objects and testing the method
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 218)
book2 = Book("1984", "George Orwell", 328)

book1.description()
book2.description()
```

Problem 2: Create a `Rectangle` Class

```python
# Solution for Problem 2
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# Creating Rectangle objects and testing the methods
rect1 = Rectangle(10, 5)
rect2 = Rectangle(7, 3)

print(f"Rectangle 1 - Area: {rect1.area()}, Perimeter: {rect1.perimeter()}")
print(f"Rectangle 2 - Area: {rect2.area()}, Perimeter: {rect2.perimeter()}")
```

Problem 3: Create a `Circle` Class

```python
import math

# Solution for Problem 3
class Circle:
    def __init__(self, radius):
```

```
        self.radius = radius

    def circumference(self):
        return 2 * math.pi * self.radius

# Creating Circle objects and testing the method
circle1 = Circle(5)
circle2 = Circle(10)

print(f"Circle 1 - Circumference: {circle1.circumference():.2f}")
print(f"Circle 2 - Circumference: {circle2.circumference():.2f}")
```

Problem 4: Implement a `Person` Class with Multiple Methods

```
# Solution for Problem 4
class Person:
    def __init__(self, name, age, occupation):
        self.name = name
        self.age = age
        self.occupation = occupation

    def introduce(self):
        print(f"Hi, my name is {self.name}, and I am a {self.occupation}.")

    def is_adult(self):
        return self.age >= 18

# Creating Person objects and testing the methods
person1 = Person("Alice", 25, "Engineer")
person2 = Person("Bob", 16, "Student")

person1.introduce()
print(f"Is {person1.name} an adult? {'Yes' if person1.is_adult() else 'No'}")

person2.introduce()
print(f"Is {person2.name} an adult? {'Yes' if person2.is_adult() else 'No'}")
```

Problem 5: Create a `Library` Class

```
# Solution for Problem 5
class Library:
    def __init__(self):
        self.collection = []  # List to store book objects

    def add_book(self, book):
        self.collection.append(book)
        print(f'Added "{book.title}" to the library.')

    def remove_book(self, title):
        self.collection = [book for book in self.collection if book.title !=
title]
        print(f'Removed "{title}" from the library.')

    def list_books(self):
```

```
        print("Books in the Library:")
        for book in self.collection:
            print(f'- "{book.title}" by {book.author}')

# Creating Book objects
book1 = Book("To Kill a Mockingbird", "Harper Lee", 281)
book2 = Book("The Hobbit", "J.R.R. Tolkien", 310)
book3 = Book("Pride and Prejudice", "Jane Austen", 279)

# Creating a Library and testing the methods
library = Library()
library.add_book(book1)
library.add_book(book2)
library.add_book(book3)

library.list_books()
library.remove_book("The Hobbit")
library.list_books()
```

Problem 6: Create a `BankAccount` and `CheckingAccount` Class Using Inheritance

```
# Solution for Problem 6
class BankAccount:
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New balance: {self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
            print(f"{amount} withdrawn. New balance: {self.balance}")

class CheckingAccount(BankAccount):
    def __init__(self, account_holder, balance=0, transaction_fee=5):
        super().__init__(account_holder, balance)
        self.transaction_fee = transaction_fee

    def withdraw(self, amount):
        total_amount = amount + self.transaction_fee
        if total_amount > self.balance:
            print("Insufficient funds to cover the withdrawal and transaction
fee.")
        else:
            self.balance -= total_amount
            print(f"{amount} withdrawn with a fee of {self.transaction_fee}.
New balance: {self.balance}")

# Testing CheckingAccount class
checking_account = CheckingAccount("John Doe", 1000)
```

```
checking_account.deposit(200)
checking_account.withdraw(100)
checking_account.withdraw(1100)  # Insufficient funds
```

Problem 7: Implement a `Vehicle` and `Car` Class Using Inheritance

```
# Solution for Problem 7
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Vehicle: {self.year} {self.make} {self.model}")

class Car(Vehicle):
    def __init__(self, make, model, year, fuel_type):
        super().__init__(make, model, year)
        self.fuel_type = fuel_type

    def display_info(self):
        print(f"Car: {self.year} {self.make} {self.model}, Fuel Type:
{self.fuel_type}")

# Creating Vehicle and Car objects
vehicle = Vehicle("Toyota", "Corolla", 2020)
car = Car("Tesla", "Model S", 2021, "Electric")

vehicle.display_info()
car.display_info()
```

Problem 8: Solution: Create a `Product` Class with Inheritance

```
# Base class: Product
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    # Base method to display product information
    def display(self):
        print(f"Product: {self.name}, Price: {self.price:.2f}")
# Child class: Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty_years):
        super().__init__(name, price)
        self.warranty_years = warranty_years

    # Overriding the display method
    def display(self):
        print(f"Product: {self.name}, Price: {self.price:.2f}, Warranty:
{self.warranty_years} years")
```

```
# Child class: Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    # Overriding the display method
    def display(self):
        print(f"Product: {self.name}, Price: {self.price:.2f}, Size:
{self.size}")

# Testing the classes
# Creating Electronics object
smartphone = Electronics("Smartphone", 699.99, 2)
# Creating Clothing object
tshirt = Clothing("T-shirt", 19.99, "M")

# Displaying the product information using the overridden methods
smartphone.display()  # Output: Product: Smartphone, Price: $699.99, Warranty:
2 years
tshirt.display()      # Output: Product: T-shirt, Price: $19.99, Size: M
```

Problem 9: Create a `Team` Class with Aggregation

```
# Solution for Problem 9
class Player:
    def __init__(self, name, position, number):
        self.name = name
        self.position = position
        self.number = number

    def display(self):
        print(f"{self.name}, Position: {self.position}, Number:
{self.number}")

class Team:
    def __init__(self, team_name):
        self.team_name = team_name
        self.players = []

    def add_player(self, player):
        self.players.append(player)
        print(f"Added {player.name} to the team.")

    def remove_player(self, name):
        self.players = [player for player in self.players if player.name !=
name]
        print(f"Removed {name} from the team.")

    def display_team(self):
        print(f"Team {self.team_name} Players:")
        for player in self.players:
            player.display()

# Testing Team and Player classes
player1 = Player("John", "Forward", 9)
```

```
player2 = Player("Alice", "Midfielder", 10)
player3 = Player("Bob", "Defender", 4)

team = Team("Dream Team")
team.add_player(player1)
team.add_player(player2)
team.add_player(player3)

team.display_team()
team.remove_player("Alice")
team.display_team()
```

Problem 10: Polymorphism with a `Shape` Class Hierarchy

```
# Solution for Problem 10
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

# Polymorphism in action
shapes = [Circle(), Square(), Circle()]
for shape in shapes:
    shape.draw()
```

In [ ]: