**Academic Year 2024-25 Even**

**19CSE313 – Principles of Programming Language**

**B.Tech CSE 2022-26 F Section**

**Practice Set 9 – Scala Classes and Objects**

1. Define a class **Person** with attributes name and age. Create an object of this class and print the details.

Code:

class Person(val name: String, val age: Int) {

def printDetails(): Unit = {

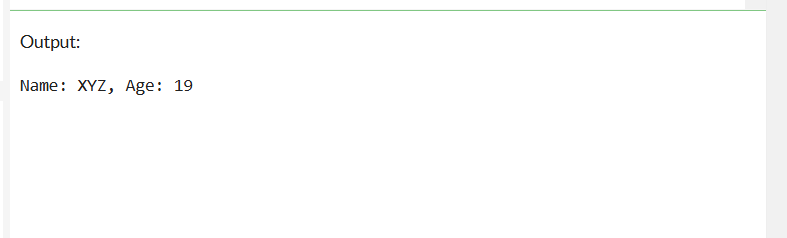
println(s"Name: $name, Age: $age")

}

}

val person = new Person("XYZ", 19)

person.printDetails()



1. Create a base class **Animal** with a method sound(). Derive a class **Dog** from Animal and override the sound() method to print "Bark".

Code:

class Animal {

def sound(): Unit = {

println("animal sound")

}

}

class Dog extends Animal {

override def sound(): Unit = {

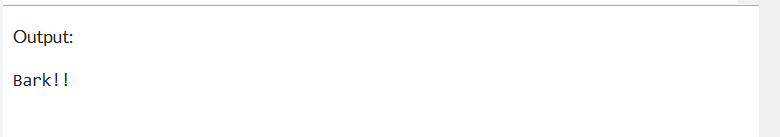
println("Bark!!")

}

}

val myDog = new Dog()

myDog.sound()



1. Explore the concept called ‘Trait’ in Scala. Explain the same with an appropriate example code. Copy and paste the screenshots of code execution
   1. Define a **trait Flyable** with a method fly(). Create a **class Bird** that extends this trait and implements the fly() method.

Traits are used to share interfaces and fields between classes. They are similar to Java 8’s interfaces. Classes and objects can extend traits, but traits cannot be instantiated and therefore have no parameters.

Defining a trait

A minimal trait is simply the keyword trait and an identifier: **trait** **HairColor**

**Code:**

trait Flyable {

def fly(): Unit

}

class Bird extends Flyable {

override def fly(): Unit = {

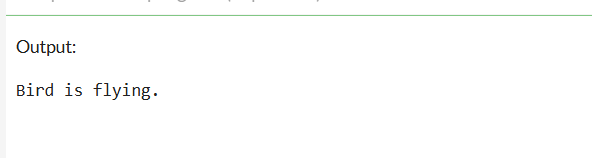
println("Bird is flying.")

}

}

val bird = new Bird()

bird.fly()



Referred to: https://docs.scala-lang.org/tour/traits.html?utm\_source=chatgpt.com

1. Explore and explain the concept of ‘ case class and case object ’ in Scala, with the help of appropriate example code.

Case objects

Case objects are to objects what case classes are to classes: they provide a number of automatically-generated methods to make them more powerful. They’re particularly useful whenever you need a singleton object that needs a little extra functionality, such as being used with pattern matching in match expressions.

Case objects are useful when you need to pass immutable messages around. For instance, if you’re working on a music player project, you’ll create a set of commands or messages like this:

**sealed** **trait** **Message**

**case** **class** **PlaySong**(name: **String**) **extends** **Message**

**case** **class** **IncreaseVolume**(amount: **Int**) **extends** **Message**

**case** **class** **DecreaseVolume**(amount: **Int**) **extends** **Message**

**case** **object** **StopPlaying** **extends** **Message**

**case class:**

// Define a case class

case class Person(name: String, age: Int)

val p1 = Person("Alice", 25) // No need for `new`

val p2 = Person("Alice", 25)

val p3 = Person("Bob", 30)

// Print objects (toString is auto-generated)

println(p1) // Output: Person(Alice,25)

// Equality check (auto-generated equals method)

println(p1 == p2) // Output: true

println(p1 == p3) // Output: false

**CASE OBJECT**

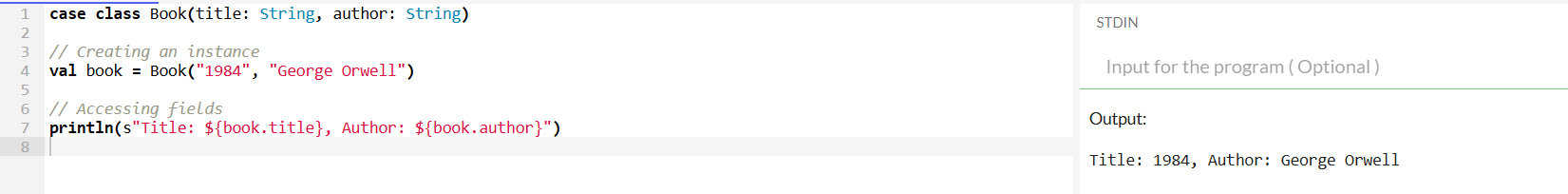
case object Earth {

def planetType(): String = "Terrestrial"

}

println(Earth.planetType()) // Output: Terrestrial

* 1. Create a **case class Book** with attributes title and author. Demonstrate how to create an instance of this case class.



Referred to: https://docs.scala-lang.org/scala3/book/domain-modeling-tools.html#case-objects

1. Create a class **BankAccount** with private attributes accountNumber and balance. Provide public methods to deposit and withdraw money, ensuring the balance cannot be negative.

Code:

class BankAccount(private val accountNumber: String, private var balance: Double) {

def deposit(amount: Double): Unit = {

if (amount > 0) {

balance += amount

println(s"Deposited $amount. New balance: $balance")

} else {

println("amount must be positive.")

}

}

def withdraw(amount: Double): Unit = {

if (amount > 0 && amount <= balance) {

balance -= amount

println(s"Withdrew $amount. Remaining balance: $balance")

} else if (amount > balance) {

println("Insufficient balance!")

} else {

println("amount must be positive.")

}

}

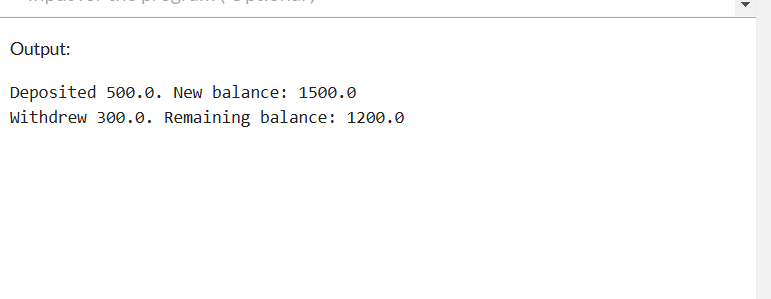
def getBalance: Double = balance

}

val myAccount = new BankAccount("1234567890", 1000.0)

myAccount.deposit(500)

myAccount.withdraw(300)



1. Explore and explain the concept of ‘abstract class’ in Scala.

Abstract classes

When you want to write a class, but you know it will have abstract members, you can either create a trait or an abstract class. In most situations you’ll use traits, but historically there have been two situations where it’s better to use an abstract class than a trait:

* You want to create a base class that takes constructor arguments
* The code will be called from Java code

A base class that takes constructor arguments:

abstract class Pet(val name: String): // Store name as a field

def greeting: String

def age: Int

override def toString = s"My name is $name, I say $greeting, and I’m $age"

class Dog(name: String, var age: Int) extends Pet(name):

val greeting = "Woof"

val d = new Dog("Fido", 1) // Use 'new' to create an object

println(d.toString)

// Abstract class

abstract class Animal {

// Abstract method (no implementation)

def sound(): Unit

// Concrete method (with implementation)

def eat(): Unit = {

println("Eating food...")

}

}

// Subclass implementing the abstract method

class Dog extends Animal {

override def sound(): Unit = {

println("Bark!!")

}

}

// Subclass implementing the abstract method

class Cat extends Animal {

override def sound(): Unit = {

println("Meow!!")

}

}

// Creating objects of subclasses

val dog = new Dog()

dog.sound() // Output: Bark!!

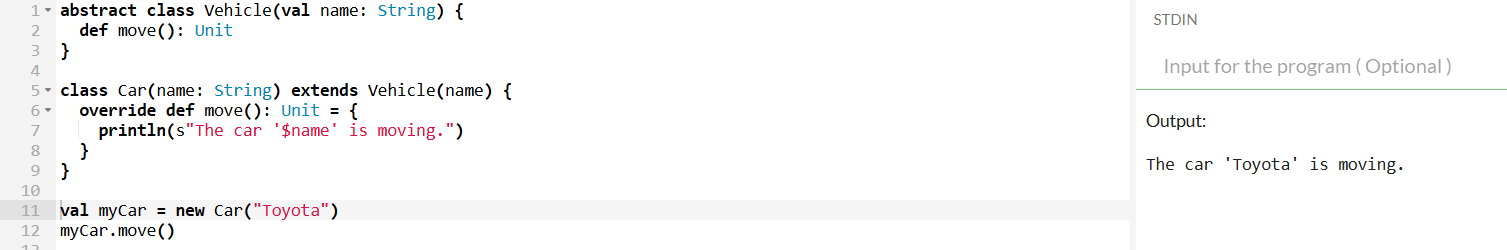
dog.eat() // Output: Eating food...

val cat = new Cat()

cat.sound() // Output: Meow!!

cat.eat() // Output: Eating food...

* 1. Create an **abstract class Vehicle** with an abstract method move(). Derive a **class Car** from Vehicle and implement the move() method.



Referred to:

<https://docs.scala-lang.org/scala3/book/domain-modeling-tools.html#abstract-classes>

1. Create a **class Counter** with a private attribute count. Provide a companion object with a method to create a new Counter instance with an initial count.

Code:

class Counter private (private var count: Int) {

def increment(): Unit = count += 1

def getCount: Int = count

}

object Counter {

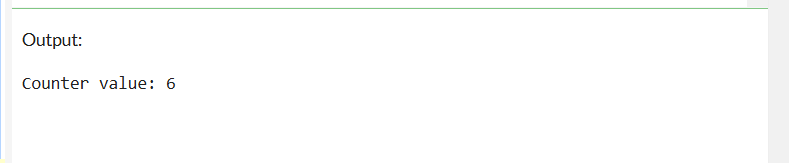
def apply(initialCount: Int): Counter = new Counter(initialCount)

}

val counter = Counter(5)

counter.increment()

println(s"Counter value: ${counter.getCount}")



**Q) Student** class with a **companion object** that manages student creation.

class Student private (val name: String, val id: Int) {

def display(): Unit = println(s"Student Name: $name, ID: $id")

}

object Student {

private var idCounter = 0 // Keeps track of student IDs

def apply(name: String): Student = {

idCounter += 1

new Student(name, idCounter) // Assigns a unique ID

}

def apply(name: String, id: Int): Student = {

new Student(name, id) // Allows custom ID assignment

}

}

// Creating Student instances

val student1 = Student("Alice") // Uses auto-generated ID

val student2 = Student("Bob", 101) // Uses custom ID

student1.display()

student2.display()

1. Create a **class Calculator** with overloaded methods add() that can take two or three integers and return their sum.

Code:

class Calculator {

def add(a: Int, b: Int): Int = a + b

def add(a: Int, b: Int, c: Int): Int = a + b + c

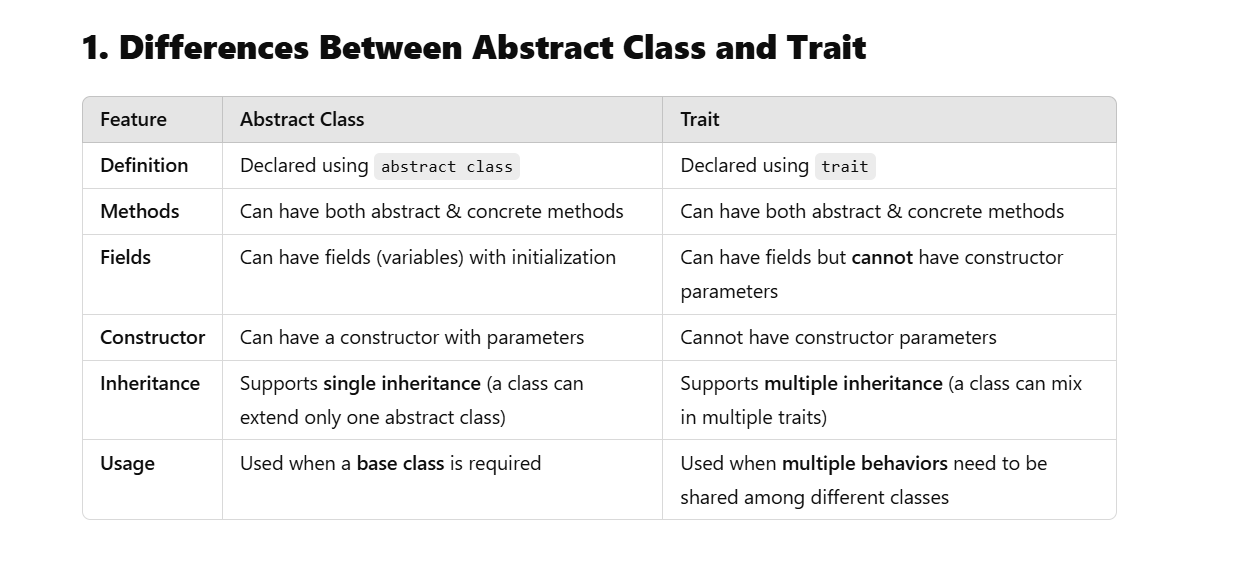
}

val calc = new Calculator()

println(calc.add(2, 3))

println(calc.add(2, 3, 4))





1. Define a **trait Shape** with a method area(). Create **classes Circle and Rectangle** that extend this trait and implement the area() method. Write a function that takes a Shape and prints its area.

Code:

trait Shape {

def area(): Double

}

class Circle(val radius: Double) extends Shape {

override def area(): Double = Math.PI \* radius \* radius

}

class Rectangle(val width: Double, val height: Double) extends Shape {

override def area(): Double = width \* height

}

def printArea(shape: Shape): Unit = {

println(s"Area: ${shape.area()}")

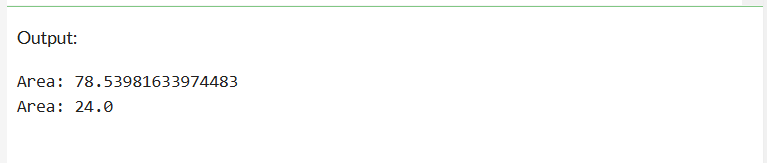
}

val circle = new Circle(5)

val rectangle = new Rectangle(4, 6)

printArea(circle)

printArea(rectangle)



1. Create a **singleton object Logger** with a method log(message: String) that prints the message to the console.

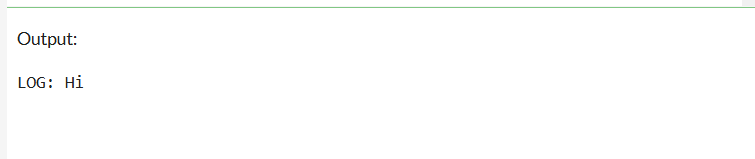
Code:

object Logger {

def log(message: String): Unit = println(s"LOG: $message")

}

Logger.log("Hi")



**Note: For the theory questions 3, 4, and 6, mention the sources referred for answer compulsorily. For programming part, if you have used LLMs/GenAI tools, ensure that those are properly cited.**