OOPS Concepts

January 2, 2025

ENCAPSULATION

[22]:

```
// Define the Encapsulation class
class Encapsulation(var number: Double) {

    // Primary constructor with one parameter
    def this(number: Double, xtype: Int) = {
        this(number)
        this.xtype = xtype
    }

    // Methods for arithmetic operations
    def add(x: Double): Unit = number += x

    def subtract(x: Double): Unit = number -= x

    def multiply(x: Double): Unit = number *= x

    def Getnumber: Double = number

    // Method to check divisibility
    def divisibility(digit: Int): Boolean = {
        number % digit == 0
    }

    // Private instance variable
    private var num: Int = 0

    // Accessor method for the private variable
    def getnum: Int = num

    // Public instance variable
    var xtype: Int = 0
}

// Main execution block
println("Inside the main function")
```

```
val obj = new Encapsulation(3000.0) // Object creation with one parameter
val obj1 = new Encapsulation(1500.0, 2) // Object creation with two parameters

// Display values and operations
println(s"Initial number in obj: ${obj.number}")
println(s"Xtype of obj1: ${obj1.xtype}")
val x = obj.getnum
println(s"Value of num through method: $x")
println(s"Checking divisibility of obj by 7: ${obj.divisibility(7)}")

obj.add(3)
println(s"Current value of number in obj after addition: ${obj.Getnumber}")
println(s"Checking divisibility of updated obj by 7: ${obj.divisibility(7)}")
```

```
Inside the main function
Initial number in obj: 3000.0
Xtype of obj1: 2
Value of num through method: 0
Checking divisibility of obj by 7: false
Current value of number in obj after addition: 3003.0
Checking divisibility of updated obj by 7: true
```

```
defined class Encapsulation
obj = Encapsulation@6738db3b
obj1 = Encapsulation@1d688776
x = 0
```

[22]: 0

Inheritance

[23]:
```
// Define a trait for aquatic behaviors
trait Aquatic {
  def info(): String = {
    "I am from the aquatic trait!"
  }

  def convertToDouble: Int => Double = i => i.toDouble
}

trait Flying {
  def details: String = {
    "I am from the flying trait"
  }
}

// Define the superclass
abstract class Creature(val typeName: String) extends Aquatic {
  // Method in the superclass
  def action(): String

  def convertToString: Int => String = i => i + ""
}

class Fish(typeName: String) extends Creature(typeName) with Aquatic { //subclass
  override def action(): String = {
    s"$typeName swims silently!"
  }
}

class Bird(typeName: String) extends Creature(typeName) { //subclass
  override def action(): String = {
    s"$typeName flies gracefully!"
  }
}

class Dolphin(typeName: String) extends Creature(typeName) {
  // Mimicking multi-level inheritance by first making abstract class inherit trait
  // and then inheriting abstract class
  override def action(): String = {
    s"$typeName jumps and plays in the water!"
  }
}

class HybridCreature(typeName: String) extends Creature(typeName) with Aquatic with Flying {
  override def action(): String = {
    s"$typeName can both swim and fly!"
  }
}

// Main execution in Apache Toree Jupyter Notebook
println("Testing Creature Behaviors:")

val myFish = new Fish("Goldfish")
println(s"Fish Info: ${myFish.info()}")
println(myFish.action())

val myBird = new Bird("Sparrow")
println(myBird.action())

val myDolphin = new Dolphin("Bottlenose Dolphin")
println(myDolphin.action())
println(myDolphin.info())

val myHybrid = new HybridCreature("Penguin")
println(myHybrid.details)
println(myHybrid.action())
println(s"Converted Value: ${myHybrid.convertToDouble(5)}")
println(s"Converted String: ${myHybrid.convertToString(42)}")

// Verifying instances
println(s"Is myFish a Creature? ${myFish.isInstanceOf[Creature]}")
println(s"Is myBird a Creature? ${myBird.isInstanceOf[Creature]}")
```

```
Testing Creature Behaviors:
Fish Info: I am from the aquatic trait!
Goldfish swims silently!
Sparrow flies gracefully!
Bottlenose Dolphin jumps and plays in the water!
I am from the aquatic trait!
I am from the flying trait
Penguin can both swim and fly!
Converted Value: 5.0Converted String: 42
Is myFish a Creature? true
Is myBird a Creature? true
```

```
defined trait Aquatic
defined trait Flying
defined class Creature
defined class Fish
defined class Bird
defined class Dolphin
defined class HybridCreature
myFish = Fish@71d5ea0e
myBird = Bird@5d170913
myDolphin = Dolphin@4ec87335
myHybrid = HybridCreature@ae1b3fc
```

[23]:

```
HybridCreature@ae1b3fc
```

Abstraction

[24]:

```
// Define an abstract class for appliances
abstract class Appliance {
    // Protected variables can be used in the subclasses
    protected val washerCapacity = 7.0
    protected val fridgeCapacity = 300.0
    protected val microwavePower = 800.0

    // Private variables can be used only within the class
```

```scala
    private val privateInfo: String = "This is a private variable in Appliance class"

    // Public by default
    var applianceType: String = "Generic Appliance"
    var energyRating: Int
    var brand: String

    def printPrivateInfo: String = s"$privateInfo"
    def printWasherCapacity: String = s"$washerCapacity"

    def efficiency(): Double // Abstract methods
    def setEnergyRating(rating: Int): Unit
    def describe(): String
}

class WashingMachine(var energyRating: Int, var brand: String) extends Appliance {
    def efficiency(): Double = {
        energyRating / washerCapacity
    }

    def setEnergyRating(rating: Int): Unit = energyRating = rating

    def describe(): String = s"Washing Machine of brand $brand with energy rating $energyRating"
}

class Refrigerator(var energyRating: Int, var brand: String) extends Appliance {
    def efficiency(): Double = {
        energyRating / fridgeCapacity
    }

    def setEnergyRating(rating: Int): Unit = energyRating = rating

    def describe(): String = s"Refrigerator of brand $brand with energy rating $energyRating"
}

class Microwave(var energyRating: Int, var brand: String) extends Appliance {
    def efficiency(): Double = {
        energyRating / microwavePower
    }

    def setEnergyRating(rating: Int): Unit = energyRating = rating
    def describe(): String = s"Microwave of brand $brand with energy rating␣
  ↪$energyRating"
}

// Main execution in Apache Toree Jupyter Notebook
println("Testing Appliance Behaviors:")

val lgWasher = new WashingMachine(5, "LG")
println(s"Appliance Type: ${lgWasher.applianceType}")
println(s"Private Info: ${lgWasher.printPrivateInfo}")
println(s"Washer Capacity: ${lgWasher.printWasherCapacity}")
println(s"Description: ${lgWasher.describe()}")
println(s"Efficiency: ${lgWasher.efficiency()}")

val samsungFridge = new Refrigerator(3, "Samsung")
println(s"Description: ${samsungFridge.describe()}")
println(s"Efficiency: ${samsungFridge.efficiency()}")

val panasonicMicrowave = new Microwave(4, "Panasonic")
println(s"Description: ${panasonicMicrowave.describe()}")
println(s"Efficiency: ${panasonicMicrowave.efficiency()}")
```

```
Testing Appliance Behaviors:
Appliance Type: Generic Appliance
Private Info: This is a private variable in Appliance class
Washer Capacity: 7.0
Description: Washing Machine of brand LG with energy rating 5
Efficiency: 0.7142857142857143
Description: Refrigerator of brand Samsung with energy rating 3
Efficiency: 0.01
Description: Microwave of brand Panasonic with energy rating 4
Efficiency: 0.005
```

```
defined class Appliance
defined class WashingMachine
defined class Refrigerator
defined class Microwave
lgWasher = WashingMachine@7becd62c
samsungFridge = Refrigerator@68cabd7d
panasonicMicrowave = Microwave@71ea8691
```

[24]:

```
Microwave@71ea8691
```

Polymorphism

[25]:

```
implicit class EnhancedDouble(val x: Double) {
  def rangeDescription(): String = {
    if (x >= 0.0 && x <= 50.0)
```

```
        "Low range"
    else if (x > 50.0 && x <= 200.0)
        "Medium range"
    else if (x > 200.0)
        "High range"
    else
        "Negative value"
  }
}

abstract class Measurement {
    def calculate(params: Double*): Double // Abstract method for variable␣
 ↪number of parameters
    def validate(params: Double*): Boolean
}

class Temperature extends Measurement {
    override def calculate(params: Double*): Double = {
        if (params.length != 1) {
            println("Provide one temperature value")
            return -1.0
        }
        val temperature = params(0)
        println(s"Temperature measured: $temperature")
        temperature
    }

    override def validate(params: Double*): Boolean = {
        params.forall(_ >= -273.15) // Ensure temperature is above absolute zero
    }
}

class Distance extends Measurement {
    override def calculate(params: Double*): Double = {
        if (params.length != 2) {
            println("Provide start and end points for distance")
            return -1.0
        }
        val distance = Math.abs(params(1) - params(0))
        println(s"Distance calculated: $distance")
        distance
    }

    override def validate(params: Double*): Boolean = {
        params.forall(_ >= 0)
    }
}
```

```scala
class Weight extends Measurement {
    override def calculate(params: Double*): Double = {
        if (params.length != 1) {
            println("Provide one weight value")
            return -1.0
        }
        val weight = params(0)
        println(s"Weight measured: $weight")
        weight
    }

    override def validate(params: Double*): Boolean = {
        params.forall(_ > 0)
    }
}

// Main execution in Apache Toree Jupyter Notebook
println("Measuring Temperature:")
val temperatureSensor = new Temperature
val temp = temperatureSensor.calculate(25.0)
println(s"Temperature: $temp, Description: ${temp.rangeDescription()}")

println("Calculating Distance:")
val distanceSensor = new Distance
val dist = distanceSensor.calculate(10.0, 50.0)
println(s"Distance: $dist")

println("Measuring Weight:")
val weightSensor = new Weight
val weight = weightSensor.calculate(75.0)
println(s"Weight: $weight, Description: ${weight.rangeDescription()}")
```

```
Measuring Temperature:
Temperature measured: 25.0
Temperature: 25.0, Description: Low range
Calculating Distance:
Distance calculated: 40.0
Distance: 40.0
Measuring Weight:
Weight measured: 75.0
Weight: 75.0, Description: Medium range
```

```
defined class EnhancedDouble
defined class Measurement
defined class Temperature
defined class Distance
defined class Weight
```

```
temperatureSensor = Temperature@73469108
temp = 25.0
distanceSensor = Distance@22503229
dist = 40.0
weightSensor = Weight@5d6f6744
weight = 75.0
```

[25]:

```
75.0
```

Companion

[26]:

```
trait Atom {
    def atomicNumber: Int
    def symbol: String
}

class Nucleus(protons: Int, neutrons: Int) {
    def massNumber: Int = protons + neutrons
}

case class Element(name: String, atomicNumber: Int, symbol: String) // Case␣
 ↪class

object Element {  // Companion object
    def set(name: String, atomicNumber: Int, symbol: String): Element = {
        new Element(name, atomicNumber, symbol)
    }
    def output(elements: Element): String = s"Name: ${elements.name}, Atomic␣
 ↪Number: ${elements.atomicNumber}"
}

// Main execution in Apache Toree Jupyter Notebook
println("Testing Atomic Properties:")

val hydrogen = Element("Hydrogen", 1, "H")
println(Element.output(hydrogen))

val hydrogenNucleus = new Nucleus(1, 0)
println(s"Hydrogen Nucleus Mass Number: ${hydrogenNucleus.massNumber}")

val hydrogenAtom = new Atom { // Anonymous class
    def atomicNumber: Int = hydrogen.atomicNumber
    def symbol: String = hydrogen.symbol
}
println(s"Atom: ${hydrogenAtom.symbol}, Atomic Number: ${hydrogenAtom.
 ↪atomicNumber}")
```

```
Testing Atomic Properties:
```

```
Name: Hydrogen, Atomic Number: 1
Hydrogen Nucleus Mass Number: 1
Atom: H, Atomic Number: 1
```

```
defined trait Atom
defined class Nucleus
defined class Element
defined object Element
hydrogen = Element(Hydrogen,1,H)
hydrogenNucleus = Nucleus@7570f357
hydrogenAtom = $anon$1@774fa3f9
```

[26]:
```
$anon$1@774fa3f9
```

[ ]:
```

```