Objects and Classes
Question: In a package named "execution" create a Scala class named "VoteCounter" with the following:

- A constructor that takes a List of Strings representing the possible options for voting
- A method named addVotes that takes a String and an Int as parameters and returns Unit. This method adds this many votes to the option specified by the string
  - If the option is not in the list given in the constructor, the votes should be ignored
  - Example: voteCounter.addVotes("Boaty McBoatface", 1000) will add 1000 votes to this option if it was listed in the constructor call
- A method named getVotes that takes an option as a String and returns the total number of votes for this option as an Int
  - If the input is not a valid option, return 0

Testing: In a package named "tests" create a Scala class named "TestVoting" as a test suite that tests all the functionality listed above
Objects

- State / Variables
  - Objects store their state in variables
  - [Vocab] Often called fields, member variables, or instance variable

- Behavior / Functions
  - Objects contains functions that can depend on its state
  - [Vocab] When a function is part of an object it's called a method
Object With State

```scala
object ObjectWithState {

  // State of the object
  var x: Int = 10
  var y: Int = 7

  // Behavior of the object
  def doubleX(): Unit = {
    this.x *= 2
  }
}
```

- Any variable outside of all methods is part of the state of that object
- Keyword `this` stores a reference to the enclosing object
- Use `this.<variable_name>` to access state from within the object
Object With State

```scala
object ObjectWithState {

  // State of the object
  var x: Int = 10
  var y: Int = 7

  // Behavior of the object
  def doubleX(): Unit = {
    this.x *= 2
  }
}
```

- Declare variables using `var` if the value can change
- Declare variables using `val` to prevent the value from changing
  - Changing the value of a variable declared with `val` will cause an error
Object With State

```java
object ObjectWithState {
  // State of the object
  var x: Int = 10
  var y: Int = 7

  // Behavior of the object
  def doubleX(): Unit = {
    this.x *= 2
  }
}
```

- The variables defining the state of an object have many different names
  - Instance variables
  - Member variables
  - Fields
  - State variables
Object With State

object ObjectWithState {

    // State of the object
    var x: Int = 10
    var y: Int = 7

    // Behavior of the object
    def doubleX(): Unit = {
        this.x *= 2
    }
}

object ObjectMain {

    def main(args: Array[String]): Unit = {
        ObjectWithState.doubleX()
        println(ObjectWithState.x)
    }
}

• Any code with access to an object can also access its state/behavior with the dot notation

• Can also change the state of an object
Every **value** in Scala is an **object**
Classes

- Classes are templates for creating objects with similar state and behavior
  - Objects are instantiated from classes using the keyword \texttt{new}

- Used to create many objects
  - Each object can have a different state
  - Each has its own copies of the instance variables
Define a class to represent an item in a store

```scala
class Item(val description: String, var price: Double) {
  var timesPurchased: Int = 0

  def purchase(): Unit = {
    this.timesPurchased += 1
  }

  def onSale(): Unit = {
    this.price *= 0.8
  }
}
```

- Define a class to represent an item in a store
• State and behavior is defined the same way as objects

• We define one state variable to track the number of times this item was purchased along with a method/behavior to purchase an item

• We define more behavior to mark an item as on sale by reducing its price by 20%
Classes

- Classes also contain special methods called constructors
- This method is called when a new object is created using this class
- Any code calling the constructor can use its parameters to set the initial state of the created object
- [Scala] All constructor parameters become member variables
  - Use var in the constructor if the state can be change
Classes

object ItemMain {

  def printPrice(item: Item): Unit = {
    println("Current price of "+ item.description +" is: "+ item.price)
  }

  def main(args: Array[String]): Unit = {
    val cereal: Item = new Item("cereal", 3.0)
    val milk: Item = new Item("milk", 2.0)

    // Change state using behavior
    cereal.purchase()
    cereal.onSale()
    cereal.purchase()

    println(cereal.description + " has been purchased "+ cereal.timesPurchased + " times")
    printPrice(cereal)

    // Change state directly
    milk.price = 1.5

    printPrice(milk)
  }
}

• Call a constructor using the new keyword
• The constructor returns a reference to the created class of the type of the class
Classes

object ItemMain {

  def printPrice(item: Item): Unit = {
    println("Current price of " + item.description + " is: $" + item.price)
  }

  def main(args: Array[String]): Unit = {

    val cereal: Item = new Item("cereal", 3.0)
    val milk: Item = new Item("milk", 2.0)

    // Change state using behavior
    cereal.purchase()
    cereal.onSale()
    cereal.purchase()

    println(cereal.description + " has been purchased " + cereal.timesPurchased + " times")
    printPrice(cereal)

    // Change state directly
    milk.price = 1.5

    printPrice(milk)
  }
}

• We have two different objects of type Item
• cereal and milk have their own copies of each instance variable
References

• Every class you create will be passed by reference

• Also data structure (List, Map, Array) and other built-in classes

• Pass-by-reference means that a copy is not made when a variable is assigned a value
References

- increasePrice returns Unit, yet it is able to modify an item
- cereal and cereal2 "refer" to the same object
  - Changes made to one will change both variables
Classes

• Method parameters, including constructors, can have default values

• Any missing arguments are set to the default value

```scala
class PhysicsVector(var x: Double = 0.0, var y: Double = 0.0, var z: Double = 0.0) {
  override def toString: String = {
    "(" + x + ", " + y + ", " + z + ")"
  }
}

val vector: PhysicsVector = new PhysicsVector(4.0, -3.5, 0.7) // (4.0, -3.5, 0.7)
val vector2: PhysicsVector = new PhysicsVector(-6.0) // (-6.0, 0.0, 0.0)
val vector3: PhysicsVector = new PhysicsVector() // (0.0, 0.0, 0.0)
```
Classes

- Can define a toString method to print an object with custom formatting

```scala
class PhysicsVector(var x: Double = 0.0, var y: Double = 0.0, var z: Double = 0.0) {
  override def toString: String = {
    "(" + x + ", " + y + ", " + z + ")"
  }
}
```

val vector: PhysicsVector = new PhysicsVector(4.0, -3.5, 0.7) // (4.0, -3.5, 0.7)
val vector2: PhysicsVector = new PhysicsVector(-6.0) // (-6.0, 0.0, 0.0)
val vector3: PhysicsVector = new PhysicsVector() // (0.0, 0.0, 0.0)
• previousLocation and location are the same object!!
  • Changing location will change previousLocation!

• Create a new PhysicsVector for previousLocation or copy x, y, z one at a time

```scala
def updateObject(dynamicObject: DynamicObject, deltaTime: Double, magnitudeOfGravity: Double): Unit = {
  dynamicObject.previousLocation = dynamicObject.location
  // ... rest of the method
}
```
Classes

• Int, Double, Boolean, List, Array, Map
• Are all classes
• We use these classes to create values

```
var list: List[Int] = List(2, 3, 4)
```

• Create objects by calling the constructor for that class
• List is setup in a way that we don't use `new`
• For our classes we will use the `new` keyword
A Note on Access Modifiers

- Determine who (which classes/objects) can alter state and control behavior of an object.

- Access modifiers are controversial.

- Communities around different languages cannot agree on these.
Access Modifiers

• If you're familiar with **Java** you're familiar with these
  • public / private / protected
  • default is package private

• In **Scala**
  • private / protected
  • default is public

• In **Python**
  • No access modifiers
  • Everything is public

• In **JavaScript**
  • No access modifiers
  • Everything is public
  • Can create work-arounds to simulate private variables
Common in some languages to make all member variables private

- Java
- C++

State is never accessed directly from outside the object

- Use accessor (getter) and mutator (setter) methods instead

```java
package oop_classes;

public class AccessModifiers{

    // NOTE: This is Java code

    private int x;

    public int getX(){
        return this.x;
    }

    public void setX(int x){
        this.x = x;
    }

}  // end class AccessModifiers
```
Question: In a package named "execution" create a Scala class named "VoteCounter" with the following:

- A constructor that takes a List of Strings representing the possible options for voting
- A method named `addVotes` that takes a String and an Int as parameters and returns Unit. This method adds this many votes to the option specified by the string
  - If the option is not in the list given in the constructor, the votes should be ignored
  - Example: `voteCounter.addVotes("Boaty McBoatface", 1000)` will add 1000 votes to this option if it was listed in the constructor call
- A method named `getVotes` that takes an option as a String and returns the total number of votes for this option as an Int
  - If the input is not a valid option, return 0

Testing: In a package named "tests" create a Scala class named "TestVoting" as a test suite that tests all the functionality listed above