Objects and Classes

Lecture Question

Question: In a package named "rhymes" create a Scala **class** named "Word" with the following:

- A constructor that takes a **val**ue of type List of Strings representing the sounds of the word with each sound as a separate element in the list
  - For more information on sounds, see the Rhyming Dictionary HW handout
- A method named "alliterationLength" that takes a Word as a parameter and returns an Int. This method returns the number of sounds at the beginning of the 2 words that match
  - Ex. The Lists of sounds
    - List("K", "AH0", "L", "AE1", "M", "AH0", "T", "AH0", "S") and
    - List("K", "AH0", "L", "IH1", "P", "S", "OW2")
    - have an alliteration length of 3 since the first 3 sounds are identical

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

Objects

Objects have State and Behavior

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

```
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 the 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 a value declared with val will cause an error

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
  }

}
```

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

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
  }

}
```

```scala
object ObjectMain {

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

}
```

- Any code with access to an object can also access it's 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 **new**
- Used to create many objects
  - Each object can have a different state
  - Each has its own copies of the state variables

Classes

```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

Classes

```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
  }

}
```

- 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

```
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
  }

}
```

- 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 state variables
  - Use **var** in the constructor if the state variable can 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

```scala
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

Classes

- Int, Double, Boolean, List, Array, Map
  - Are all classes
  - We use these classes to create objects

```
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

Testing Classes Demo

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 + ")"
  }

}
```

```scala
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

```
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)
```

Lecture Question

Question: In a package named "rhymes" create a Scala **class** named "Word" with the following:

- A constructor that takes a **val**ue of type List of Strings representing the sounds of the word with each sound as a separate element in the list
  - For more information on sounds, see the Rhyming Dictionary HW handout
- A method named "alliterationLength" that takes a Word as a parameter and returns an Int. This method returns the number of sounds at the beginning of the 2 words that match
  - Ex. The Lists of sounds
    - List("K", "AH0", "L", "AE1", "M", "AH0", "T", "AH0", "S") and
    - List("K", "AH0", "L", "IH1", "P", "S", "OW2")
    - have an alliteration length of 3 since the first 3 sounds are identical

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

Lecture Question

- **Notes and implications**:
    - A Word with an empty list of sounds always has iteration length 0 with other words
    - An entire word can be part of the alliteration of another word
        - Ex. "home"/List("HH", "OW1", "M") and "homework"/List("HH", "OW1", "M", "W", "ER2", "K"))
    - Since alliterationLength is part of the behavior of the new objects created from the Word class, it has access to the list of sounds from the constructor call by using the keyword "this"
        - Compare "this" list of sounds with the sounds of the input of the method
    - To iterate over 2 Lists simultaneously you can iterate over the indices of the lists and call the apply method on each list to access the appropriate elements on each iteration of your loop
    - The list of sounds in the constructor does not have a defined name
        - DO NOT access the list of sounds in your tests. In all likelihood, my code on AutoLab will have a different name than the one you chose and your tests will cause errors