Model of Execution
**Question**: in a package named "execution", implement the following classes:

- **class Valuable**
  - Constructor has no parameter

- **class Trader**
  - Constructor takes one variable as a parameter named "item" of type Valuable (use var)
  - This class represents someone with a valuable item they are willing to trade. Each trader can only own 1 item at a time which is stored in the "item" state variable

- **class TradeAgreement**
  - Constructor takes 2 objects of type Trader
  - One method named "executeTrade" that take no parameters and returns Unit
    - When this method is called, swap the items belonging to the 2 traders from the constructor (ie. When a TradeAgreement is created, 2 people are agreeing to a trade. When executeTrade is called, they physically trade those items)
    - The agreement can only be executed once. If this method is called more than once for a single agreement, additional trades are not made (ie. The people do not trade back to their original items if this method is called twice)

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

**Note**: The default behavior of == is to compare by reference. Comparing values of a type your created will return true only if the 2 values refer to the same object
More Memory Examples

- Multiple Objects on the heap
- Multiple frames on the stack
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}

More Memory Examples

• Multiple Objects on the heap
• Start program with command line args on the stack

• Ask OS for heap space for 1 Bird

```scala
class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}

def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if (bird.inDanger()) {
    val action: String = "Panic!"
  } else {
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if (box.inDanger()) {
    action = "Stay in the boat"
  }
  println(action)
}
```
• Declare variable action
• Add to stack
<table>
<thead>
<tr>
<th>RAM @42976</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object of type Bird</td>
</tr>
<tr>
<td>-timesHelpful value:0</td>
</tr>
<tr>
<td>-timesChecked value:0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>name:bird, value:42976</td>
</tr>
<tr>
<td>name:action, value:&quot;Nothing&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;new stack frame&gt;</th>
</tr>
</thead>
</table>

```scala
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}
```

```scala
class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}
```

```scala
class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
```

- Call method
- Create new stack frame
- increment timesChecked
def main(args: Array[String]): Unit = {
    val bird: Bird = new Bird()
    var action: String = "Nothing"
    if(bird.inDanger()){
        val action: String = "Panic!"
    }else{
        val action: String = "Check bird"
    }
    println(action)
    val box: Box = new Box(bird, new Bird())
    if(box.inDanger()){
        action = "Stay in the boat"
    }
    println(action)
}

class Bird {
    val timesHelpful: Int = 0
    var timesChecked: Int = 0
    def inDanger(): Boolean = {
        timesChecked += 1
        true
    }
}

class Box(val bird1: Bird, val bird2: Bird) {
    def inDanger(): Boolean = {
        bird1.inDanger() && bird2.inDanger()
    }
}

- Destroy stack frame
- Enter if block
- Declare value action
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"

  if (bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }

  println(action)

  val box: Box = new Box(bird, new Bird())

  if (box.inDanger()){
    action = "Stay in the boat"
  }

  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0

  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
Print the string "Nothing"
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
• Destroy stack frame for bird2.inDanger
• Enter if block
• Find action in outer scope
• Destroy stack frame for bird2.inDanger
• Enter if block
• Find action in outer scope

def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if (bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if (box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
• Program ends
• Free all memory

```scala
def main(args: Array[String]): Unit = {
  val bird: Bird = new Bird()
  var action: String = "Nothing"
  if(bird.inDanger()){
    val action: String = "Panic!"
  }else{
    val action: String = "Check bird"
  }
  println(action)
  val box: Box = new Box(bird, new Bird())
  if(box.inDanger()){
    action = "Stay in the boat"
  }
  println(action)
}

class Bird {
  val timesHelpful: Int = 0
  var timesChecked: Int = 0
  def inDanger(): Boolean = {
    timesChecked += 1
    true
  }
}

class Box(val bird1: Bird, val bird2: Bird) {
  def inDanger(): Boolean = {
    bird1.inDanger() && bird2.inDanger()
  }
}
```
More Memory Examples

• Multiple frames on the stack

``` scala
def computeGeometricSum(n: Int): Int = {
  if(n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}
def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
</tbody>
</table>

- Call function
- Create new stack frame
def computeGeometricSum(n: Int): Int = {
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}

Recursive Example

Enter if block

Call function again

Create new stack frame
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
    if (n > 0) {
        var result: Int = computeGeometricSum(n - 1)
        result += n
        result
    } else {
        0
    }
}

def main(args: Array[String]): Unit = {
    val result: Int = computeGeometricSum(3)
    println(result)
}
```

- In next function call, conditional true
- New if block
- New stack frame

### RAM

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>

<Used by another program>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Repeat, repeat
- Many variables named n on the stack
- Each is in different frame so it's ok
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Conditional finally
  - false
- return 0

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:2</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:1</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:0</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
</tbody>
</table>
Recursive Example

def computeGeometricSum(n: Int): Int ={
    if(n>0) {
        var result: Int = computeGeometricSum(n - 1)
        result += n
        result
    }else{
        0
    }
}

def main(args: Array[String]): Unit = {
    val result: Int = computeGeometricSum(3)
    println(result)
}

• Assign return value to result
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if(n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Add value of the n in this stack frame to result
- result is the last expression and is returned

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:2</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:1</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>name:result, value:1</td>
</tr>
</tbody>
</table>

<Used by another program>
<Used by another program>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int ={
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Return to function call from previous frame
- Store return value in result

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:2</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>name:result, value:1</td>
</tr>
</tbody>
</table>

<Used by another program>
<Used by another program>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}
def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Add value of n from this frame.
- Repeat
### Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}
def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Add value of n from this frame..
- Repeat

### RAM

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;if block&gt;</td>
</tr>
<tr>
<td>name:result, value:3</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
</tbody>
</table>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- And repeat..
- Imagine if the original input were 1000
- This is why we use computers
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  } else {
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Value result in main method gets the last return value

<table>
<thead>
<tr>
<th>RAM</th>
<th>args</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name: result, value: 6</td>
</tr>
</tbody>
</table>

<Used by another program>

<Used by another program>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- print 6
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  if(n > 0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}
def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- Free memory
More Memory Examples

• We were close to the end of the stack on that example

• What if this were our code?

```scala
def computeGeometricSum(n: Int): Int ={
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
}

def main(args: Array[String]): Unit = {
    val result: Int = computeGeometricSum(3)
    println(result)
}
```
At this point the other program was going to return 0 and return back up the stack.

```scala
def computeGeometricSum(n: Int): Int = {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
}
def main(args: Array[String]): Unit = {
    val result: Int = computeGeometricSum(3)
    println(result)
}
```
Recursive Example

```scala
def computeGeometricSum(n: Int): Int = {
  var result: Int = computeGeometricSum(n - 1)
  result += n
  result
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

- This program keeps adding frames to the stack

<table>
<thead>
<tr>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:3</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:2</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:1</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:0</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:-1</td>
</tr>
<tr>
<td>&lt;new stack frame&gt;</td>
</tr>
<tr>
<td>name:n, value:-2</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
<tr>
<td>&lt;Used by another program&gt;</td>
</tr>
</tbody>
</table>
Recursive Example

```
val result: Int = computeGeometricSum(n - 1)
result += n
result
```

```
def main(args: Array[String]): Unit = {
val result: Int = computeGeometricSum(3)
println(result)
}
```

- STACK OVERFLOW
- Program crashes
Question: in a package named "execution", implement the following classes:

- **class Valuable**
  - Constructor has no parameter

- **class Trader**
  - Constructor takes one variable as a parameter named "item" of type Valuable (use var)
  - This class represents someone with a valuable item they are willing to trade. Each trader can only own 1 item at a time which is stored in the "item" state variable

- **class TradeAgreement**
  - Constructor takes 2 objects of type Trader
  - One method named "executeTrade" that take no parameters and returns Unit
    - When this method is called, swap the items belonging to the 2 traders from the constructor (ie. When a TradeAgreement is created, 2 people are agreeing to a trade. When executeTrade is called, they physically trade those items)
    - The agreement can only be executed once. If this method is called more than once for a single agreement, additional trades are not made (ie. The people do not trade back to their original items if this method is called twice)

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

Note: The default behavior of == is to compare by reference. Comparing values of a type your created will return true only if the 2 values refer to the same object