

# Heap Memory

# Lecture Task

## - Point of Sale: Lecture Task 2 -

**Functionality:** In the `store.model.items` package, write a class named **“Sale”** with the following functionality:

- A constructor that takes a variable of type `Double` named “percentOff” representing the percentage of the sale
  - The parameter must be declared using **var** and be named exactly “percentOff”
- A method named “updatePrice” that takes a price as a `Double` and returns the price with the sale applied

In the `store.model.items` package, write a class named **“SaleTestingItem”** with the following functionality:

- A constructor that takes a description `String` a price as a `Double` (Same as the item class)
- A method named “addSale” that takes a **reference** to a `Sale` object and returns `Unit`. This method should store the `Sale` in a data structure so it can be applied to the price later
- A method named “price” that doesn’t take any parameters and returns the price of the item as a `Double` with all sales applied

**Testing:** In the `tests` package, create a test suite named `LectureTask2` that tests this functionality.

# Stack Memory

- Only "primitive" values are stored directly on the stack
  - Double/Float
  - Int/Long/Short
  - Char
  - Byte
  - Boolean
- All other objects are stored in heap memory\*

**\*Stack and heap allocations vary by compiler and JVM implementations. With modern optimizations, we can never be sure where our values will be stored  
We'll use this simplified view so we can move on and learn Computer Science**

# Memory Heap

The stack is very structured

What if we want more dynamic memory?

```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:list, value: 2
second value of list: 3
<Used by another program>
...

- Add the list to the stack
- The list has 2 elements
  - Allocate space for 2 Ints



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:list, value: 2
second value of list: 3
name:x, value: 5
name:y, value: 12
<Used by another program>
...

- Add more values to the stack
- Create a variable named "x" of type Int and assign it the value 5
- Create a variable named "y" of type Int and assign it the value 12
- Both values go on the stack in the order they are declared



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:list, value: 2
second value of list: 3
name:x, value: 5
name:y, value: 12
<Used by another program>
...

- Create a new list with values
  - 1, 2, 3
- Reassign the variable "list" to this new list
  - **But how??**



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:list, value: 1
second value of list: 2
<b>[third value of list: 3] --- name:x, value: 5</b>
name:y, value: 12
<Used by another program>
...

## Option 1

- Expand the the list to contain the new element
- Conflicts with value "x"
- Could move all values down the stack
  - **Too slow**
  - **Violates LIFO**



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

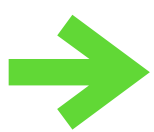


# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:x, value: 5
name:y, value: 12
name:list, value: 1
second value of list: 2
third value of list: 3
<Used by another program>
...

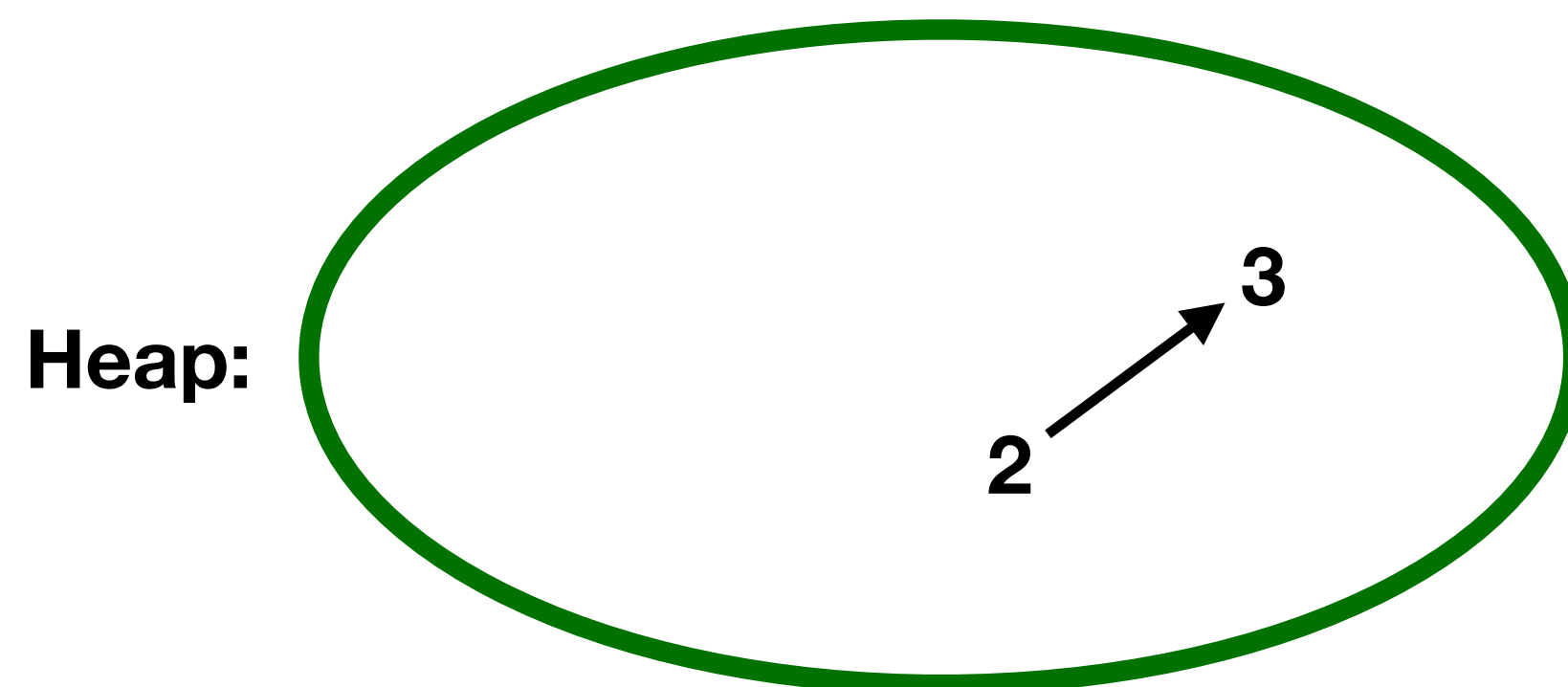
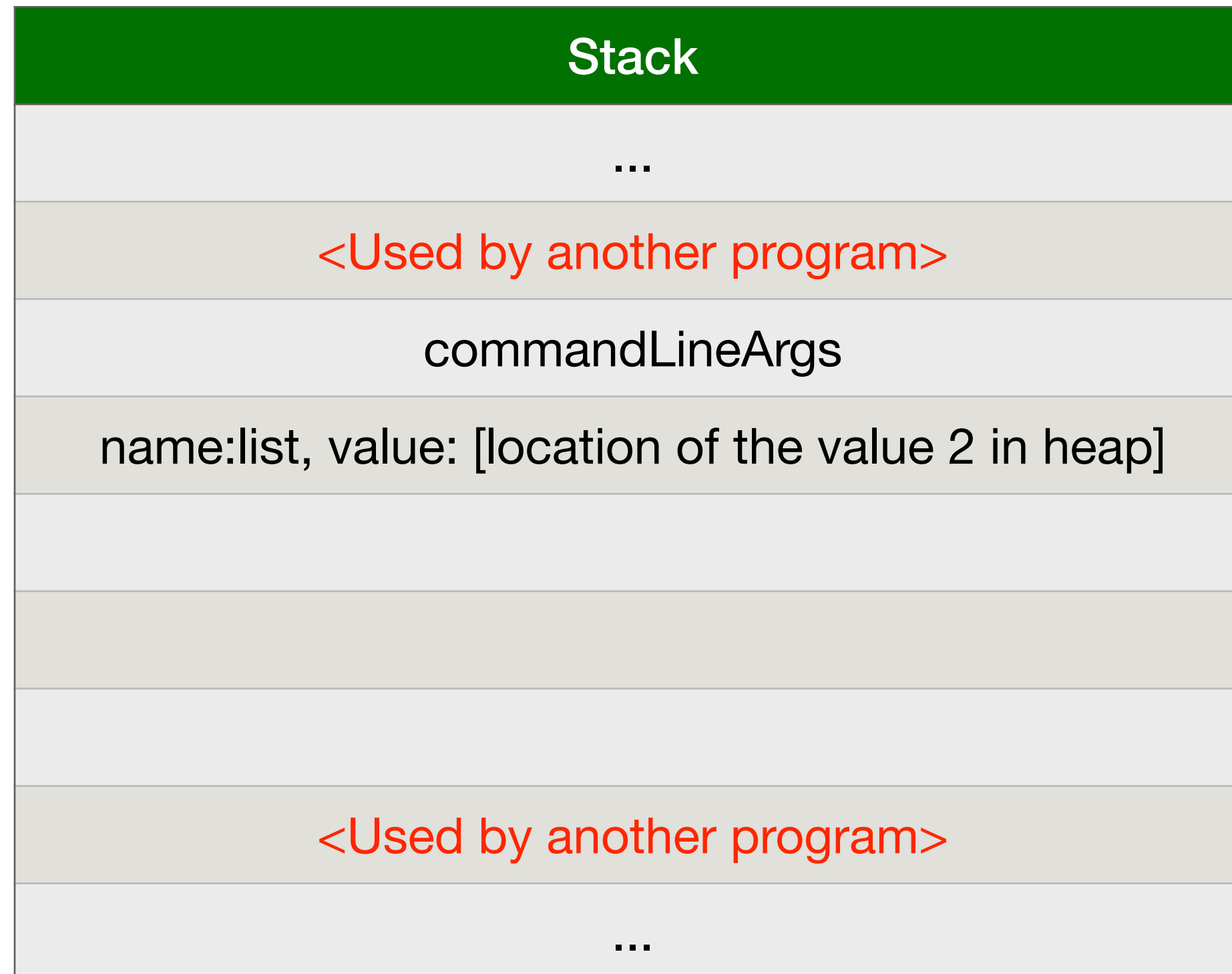
## Option 2

- Move "list" to the bottom of the stack
- Copy all values
- Delete the older copy to avoid two "list" variables in the same block
- **Too slow to copy entire list**
- **Leaves a gap in the stack**
- **Violates LIFO**



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example



## Option 3

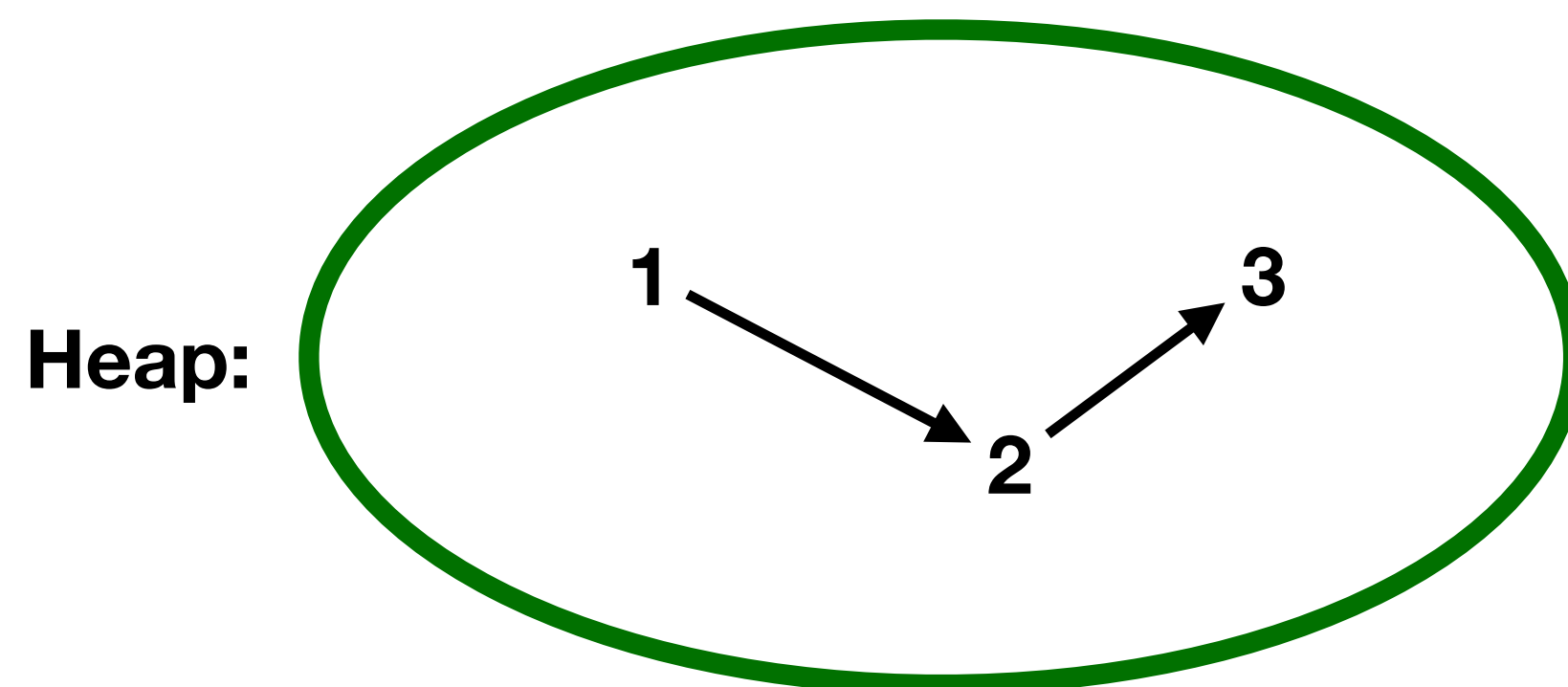
- **Allocate objects in heap memory**
- When the list is created
  - Use memory that is not part of the stack to store its values
  - Store the location, in memory, of these values in the "list" variable



```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap Example

Stack
...
<Used by another program>
commandLineArgs
name:list, value: [location of the value 1 in heap]
name:x, value: 5
name:y, value: 12
<Used by another program>
...



## Option 3

- Create a new List of Ints by adding the value 1 in heap space and have it "point" to the location of the value 2
- Reassign "list" to store the location (or reference) of the value 1

```
def main(args: Array[String]): Unit = {  
  var list: List[Int] = List(2, 3)  
  val x = 5  
  val y = 12  
  list = 1 :: list  
}
```

# Memory Heap

- Heap memory is dynamic
  - We can "ask" the OS/JVM for more heap space as needed
- Can be anywhere in RAM
  - Location is not important
  - Location can change
- Use **references** to find data
  - **Variables only store references to objects**

# References

- **Variables only store references to you objects**
  - Also data structure (List, Map, Array) and other built-in classes
- This reference tells us where in memory(the heap) to find the object
- The object itself is never stored in a variable
  - Only a reference to it
- When a method is called that takes an object, the object is **passed-by-reference**
  - A copy is never made when a variable is assigned a value
  - The method can access and change the state of the object!

# References Example

```
val player1: Player = new Player(0.0, 0.0, 10)
val player2: Player = player1

player2.move(3.0, 4.0)

checkPlayerLocation(player1, 3.0, 4.0)
checkPlayerLocation(player2, 3.0, 4.0)
```

- When a new object is created:
  - We ask the OS/JVM for enough heap memory/space to store the new object
  - Create the object and store it in this memory
  - A **reference** to that memory is returned by **new** and stored in player1

# References Example

```
val player1: Player = new Player(0.0, 0.0, 10)
val player2: Player = player1

player2.move(3.0, 4.0)

checkPlayerLocation(player1, 3.0, 4.0)
checkPlayerLocation(player2, 3.0, 4.0)
```

- When assigning an object to another variable:
  - Only the reference is assigned
  - player1 and player2 both store a reference (location in heap memory) to the same Player object
- This is the same as the assignment of parameters to arguments in a method call
  - We call this "pass-by-reference"

# References Example

```
val player1: Player = new Player(0.0, 0.0, 10)
val player2: Player = player1

player2.move(3.0, 4.0)

checkPlayerLocation(player1, 3.0, 4.0)
checkPlayerLocation(player2, 3.0, 4.0)
```

- We access an objects state and behaviour using the . dot operator
- Follows the reference to the heap
- Calling .move calls this method on the object found at that location on the heap



# References Example

```
val player1: Player = new Player(0.0, 0.0, 10)
val player2: Player = player1

player2.move(3.0, 4.0)

checkPlayerLocation(player1, 3.0, 4.0)
checkPlayerLocation(player2, 3.0, 4.0)
```

- This method call changes the state of the object
- The method returns Unit and the stack frame is destroyed
  - No memory of the method call and it didn't return a value (What's the point?)
- Since this method call changed the state of an object on the heap, the change persists

# References Example

```
val player1: Player = new Player(0.0, 0.0, 10)
val player2: Player = player1

player2.move(3.0, 4.0)

checkPlayerLocation(player1, 3.0, 4.0)
checkPlayerLocation(player2, 3.0, 4.0)
```

- Since player1 and player2 refer to the same object in heap memory:
  - player1 and player2 both "see" this movement
- There's only 1 Player object on the heap and it's state was changed during the method call
  - New objects are only created when we used the keyword new (List/Map/etc. call new internally)

# Memory Heap

Another language agnostic example

- Create an object on the heap using a class
- Modify the state of the object in a function
- The object is passed by reference

```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```

# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value: @387 (main)
<Used by another program>
...

- Create instance of ClassWithState on the heap
- Store **memory address** of the new object in data



RAM @387
...
ClassWithStateObject
-stateVar value:0
...

```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```

# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value: @387 (main)
<addToState stack frame>
name:input, value: @387 (addToState)
<Used by another program>
...

- Create a stack frame for the function call
- input is assigned the value in data
  - Which is a **memory address**



RAM @387
...
ClassWithStateObject
-stateVar value:0
...

```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```

# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value: @387 (main)
<addToState call stack frame>
name:input, value: @387 (addToState)
...
<Used by another program>
...

RAM @387
...
ClassWithStateObject
-stateVar value:1
...

```
class ClassWithState{  
    int stateVar = 0;  
}
```

- Add 1 to input.state variable
- Find the object at memory address @387
- Alter the state of the object at that address



```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```

# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value: @387 (main)
<Used by another program>
...

- Function call ends
- Destroy all data in the stack frame
- input is destroyed
- **Change to the object remains**



RAM @387
...
ClassWithStateObject
-stateVar value:1
...

```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```



# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value: @387 (main)
<Used by another program>
...

- Access data.stateVar
- Find the object at memory address @387
- Access the state of the object at that address



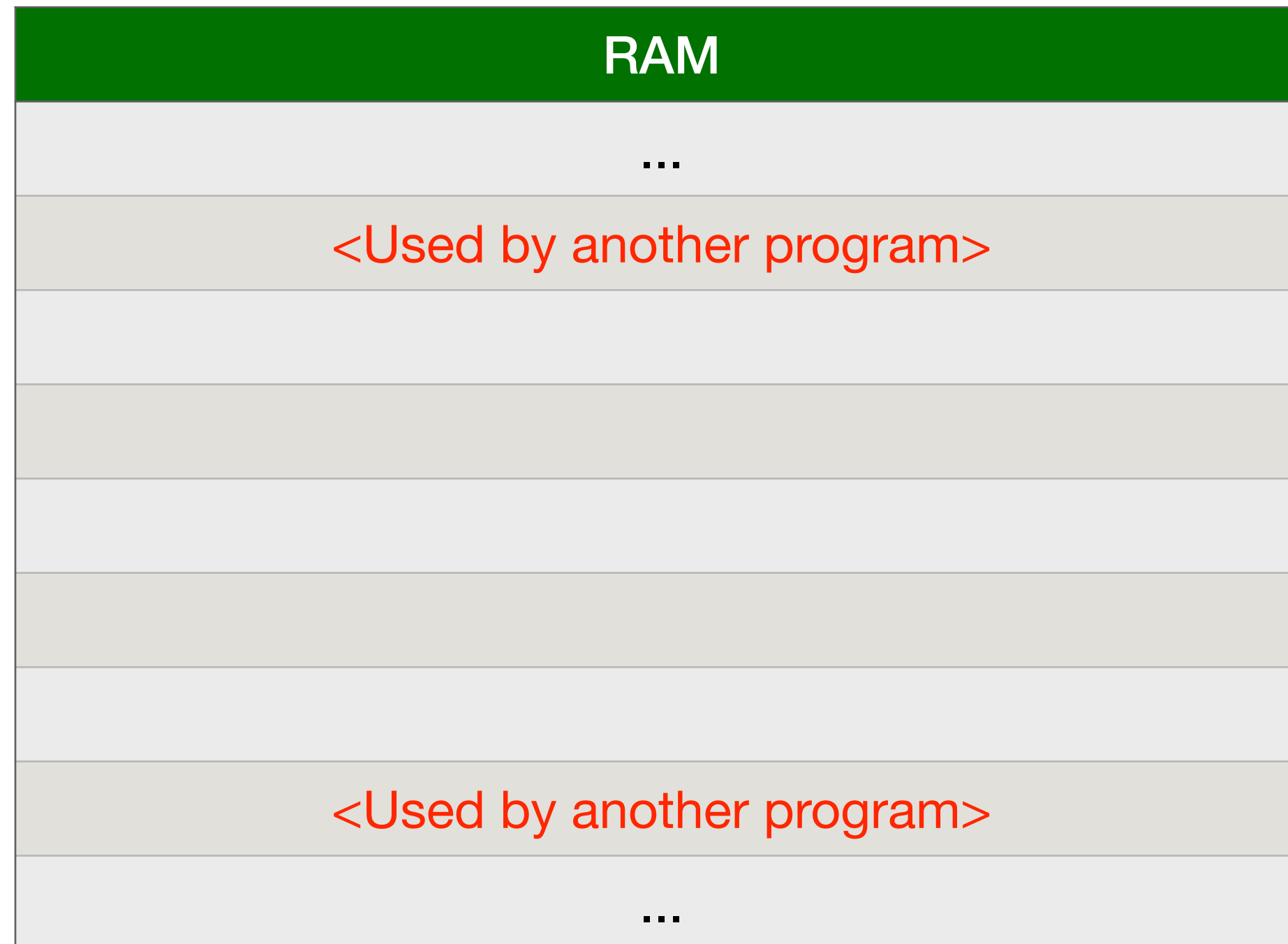
RAM @387
...
ClassWithStateObject
-stateVar value:1
...

```
class ClassWithState{  
    int stateVar = 0;  
}
```

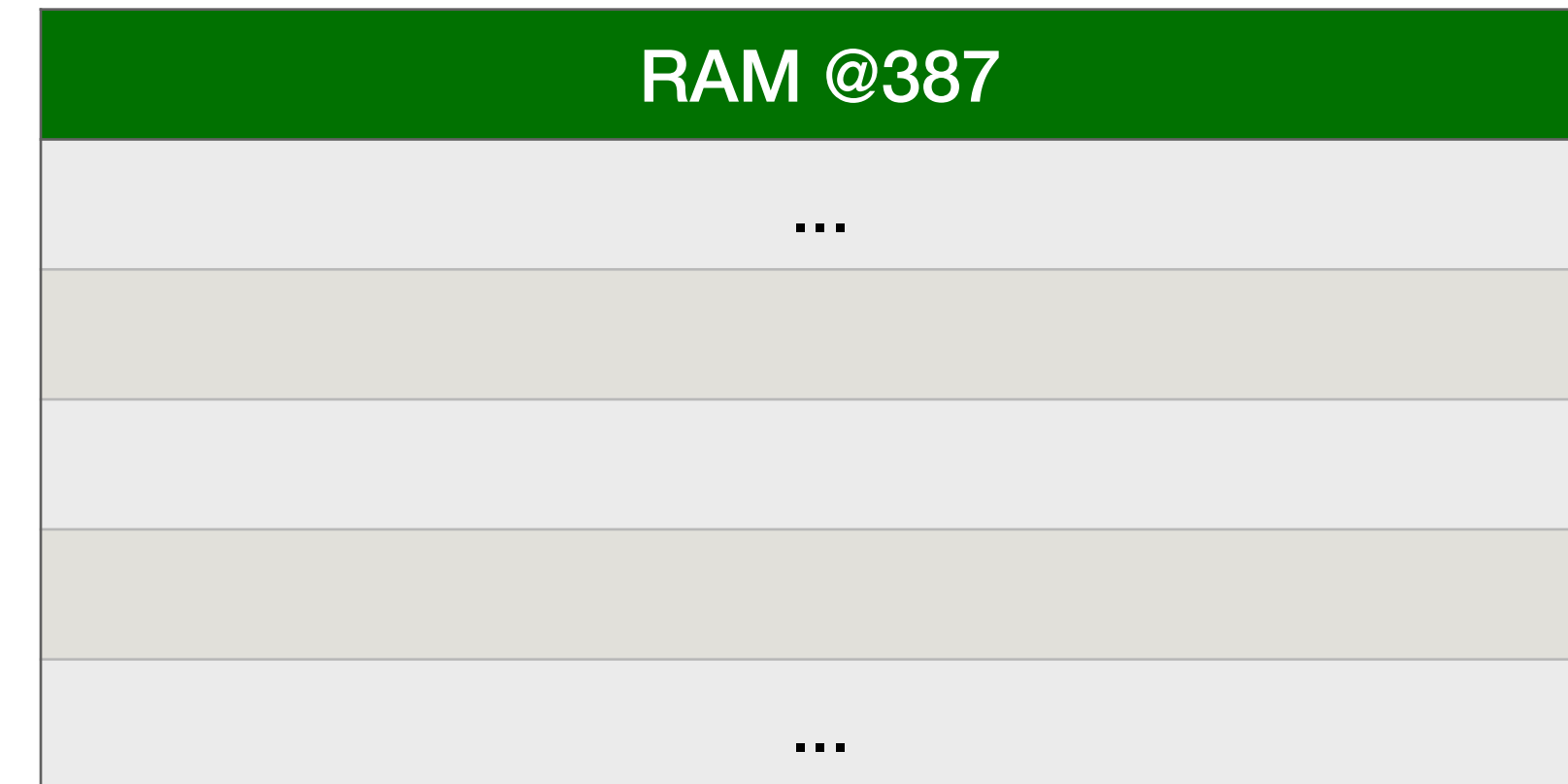
```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```



# Memory Heap Example



- All memory freed when program ends



```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
function addToState(input){  
    input.stateVar += 1  
}  
  
function main{  
    data = new ClassWithState()  
    addToState(data)  
    println(data.stateVar)  
}
```



# Lecture Task

## - Point of Sale: Lecture Task 2 -

**Functionality:** In the `store.model.items` package, write a class named “**Sale**” with the following functionality:

- A constructor that takes a variable of type `Double` named “percentOff” representing the percentage of the sale
  - The parameter must be declared using **var** and be named exactly “percentOff”
- A method named “updatePrice” that takes a price as a `Double` and returns the price with the sale applied

In the `store.model.items` package, write a class named “**SaleTestingItem**” with the following functionality:

- A constructor that takes a description `String` a price as a `Double` (Same as the item class)
- A method named “addSale” that takes a **reference** to a `Sale` object and returns `Unit`. This method should store the `Sale` in a data structure so it can be applied to the price later
- A method named “price” that doesn’t take any parameters and returns the price of the item as a `Double` with all sales applied

**Testing:** In the `tests` package, create a test suite named `LectureTask2` that tests this functionality.

# Lecture Task

## Sample Usage

```
val milk: SaleTestingItem = new SaleTestingItem("milk", 3.0)
val milkSale: Sale = new Sale(20.0)
milk.addSale(milkSale)
assert(compareDoubles(milk.price(), 2.4), milk.price())
```

## Commentary

Your SaleTestingItem method must store references to each Sale that is added

If a sale price is updated after being added to a SaleTestingItem, the price of the item should also update

You need to write a test that will check if a solution is handling references properly (no\_oob\_percent in AutoLab does not handle references properly)