

# Model of Execution

# Lecture Question

**Question:** In a package named "execution" create a Scala **class** named "Team" and a Scala **object** named "Referee".

Team will have:

- State values of type Int representing the strength of the team's offense and defense with a constructor to set these values. The parameters for the constructor should be offense then defense
- A third state variable named "score" of type Int that is not in the constructor, is declared as a **var**, and is initialized to 0

Referee will have:

- A method named "playGame" that takes two Team objects as parameters and return type Unit. This method will alter the state of each input Team by setting their scores equal to their offense minus the other Team's defense. If a Team's offense is less than the other Team's defense their score should be 0 (no negative scores)
- A method named "declareWinner" that takes two Teams as parameters and returns the Team with the higher score. If both Teams have the same score, return a new Team object with offense and defense both set to 0

# Interpretation v. Compilation

- Interpretation
  - Code is read and executed one statement at a time
- Compilation
  - Entire program is translated into another language
  - The translated code is interpreted

# Interpretation

- Python and JavaScript are interpreted languages
- Run-time errors are common
  - Program runs, but crashes when a line with an error is interpreted

**This program runs without error**

```
class RuntimeErrorExample:

    def __init__(self, initial_state):
        self.state = initial_state

    def add_to_state(self, to_add):
        print("adding to state")
        self.state += to_add

if __name__ == '__main__':
    example_object = RuntimeErrorExample(5)
    example_object.add_to_state(10)
    print(example_object.state)
```

**This program crashes with runtime error**

```
class RuntimeErrorExample:

    def __init__(self, initial_state):
        self.state = initial_state

    def add_to_state(self, to_add):
        print("adding to state")
        self.state += to_add

if __name__ == '__main__':
    example_object = RuntimeErrorExample(5)
    example_object.add_to_state("ten")
    print(example_object.state)
```

# Compilation

- Scala, Java, C, and C++ are compiled languages
- Compiler errors are common
  - Compilers will check all syntax and types and alert us of any errors (Compiler error)
  - Program fails to be converted into the target language
  - Program never runs
  - The compiler can help us find errors before they become run-time errors

## Compiles and runs without error

```
class CompilerError(var state: Int) {  
    def addToState(toAdd: Int): Unit = {  
        this.state += toAdd  
    }  
}  
  
object Main {  
    def main(args: Array[String]): Unit = {  
        val exampleObject = new CompilerError(5)  
        exampleObject.addToState(10)  
        println(exampleObject.state)  
    }  
}
```

## Does not compile. Will not run any code

```
class CompilerError(var state: Int) {  
    def addToState(toAdd: Int): Unit = {  
        this.state += toAdd  
    }  
}  
  
object Main {  
    def main(args: Array[String]): Unit = {  
        val exampleObject = new CompilerError(5)  
        exampleObject.addToState("ten")  
        println(exampleObject.state)  
    }  
}
```

# Compilation

- Compilers produce efficient code
  - While translating, the compiler "fixes" our code whenever it can
  - Compilers are very smart!
- Can even fix some major errors

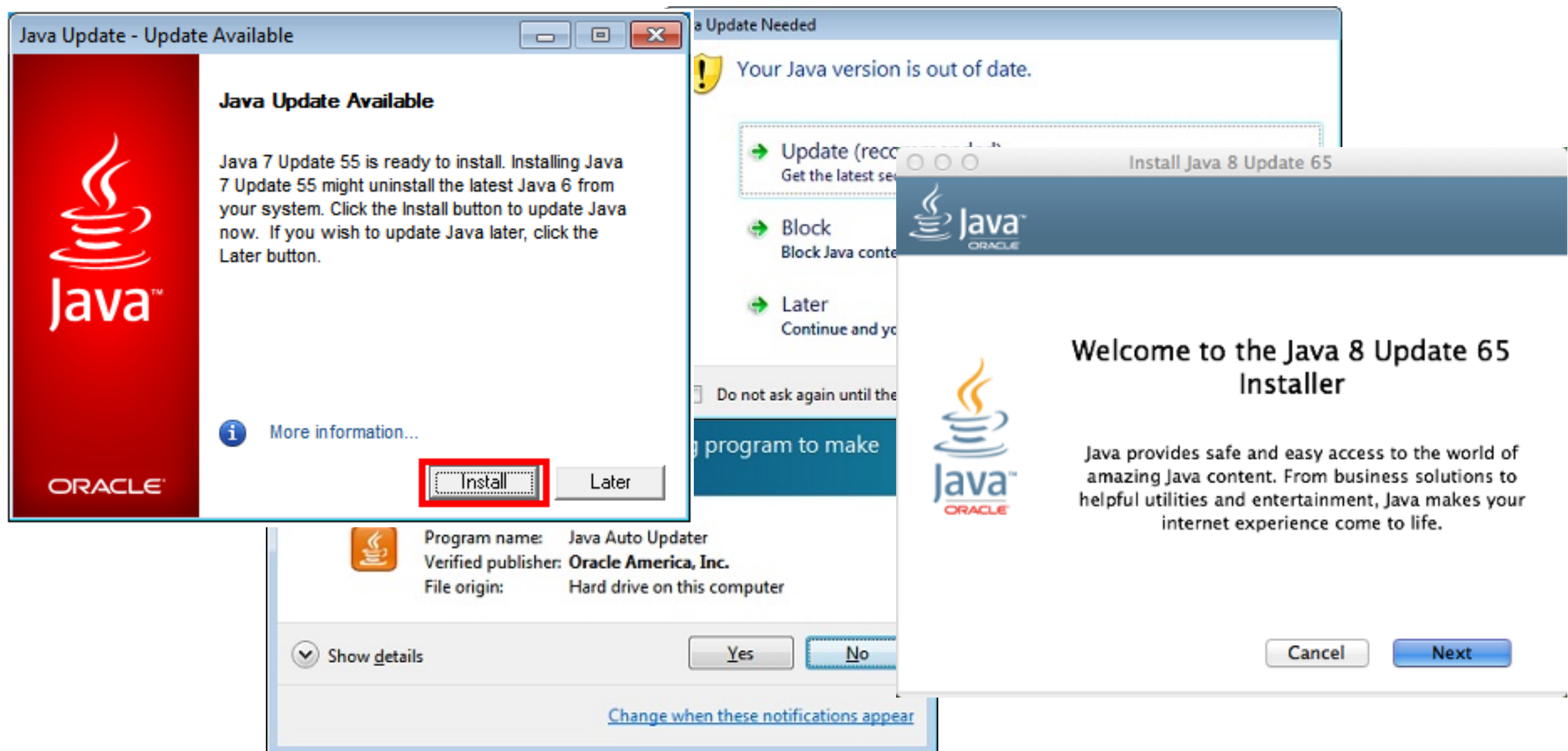
**This code runs.. forever, but it doesn't crash. Thanks compiler!**

```
object StackFlow {  
  
  def recursiveFunction(n: Int): Int = {  
    recursiveFunction(n)  
  }  
  
  def main(args: Array[String]): Unit = {  
    recursiveFunction(1)  
  }  
  
}
```

**\*If you are interested in more details about this example, search for Tail Recursion**

# Compilation - Scala

- Scala compiles to Java Byte Code
- Executed by the Java Virtual Machine (JVM)
- Installed on Billions of devices!



# Compilation - Scala

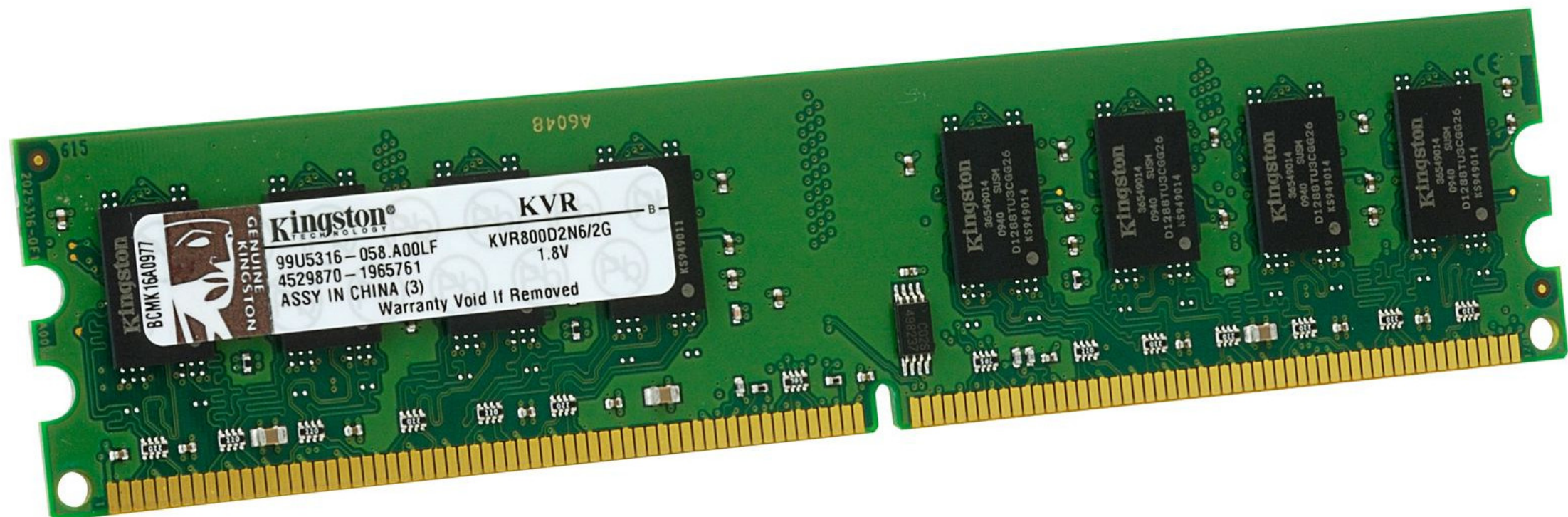
- Compiled Java and Scala code can be used in the same program
- Since they both compile to Java Byte Code
- Scala uses many Java classes
- We saw that Scala String is the same as Java String
- We'll sometimes use Java libraries in this course



# Memory

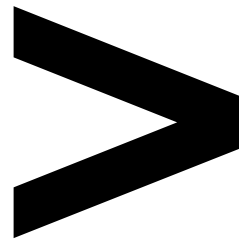
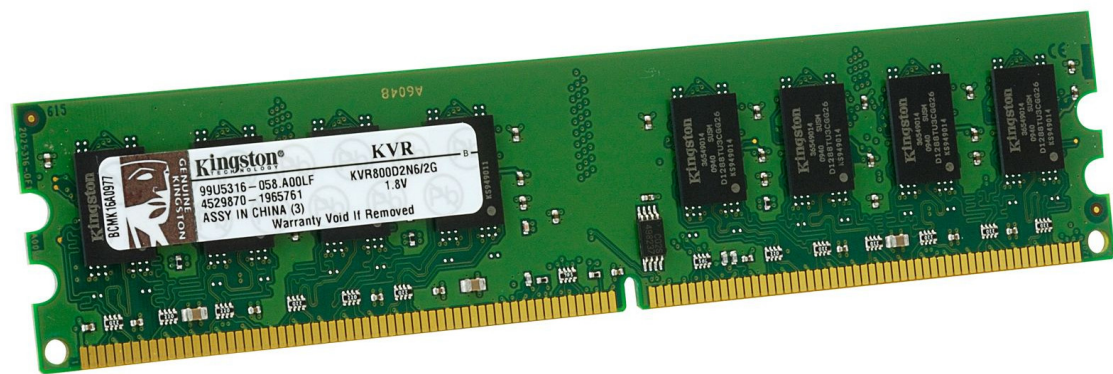
# Let's Talk About Memory

- Random Access Memory (RAM)
  - Access any value by index
  - Effectively a giant array
- All values in your program is stored here



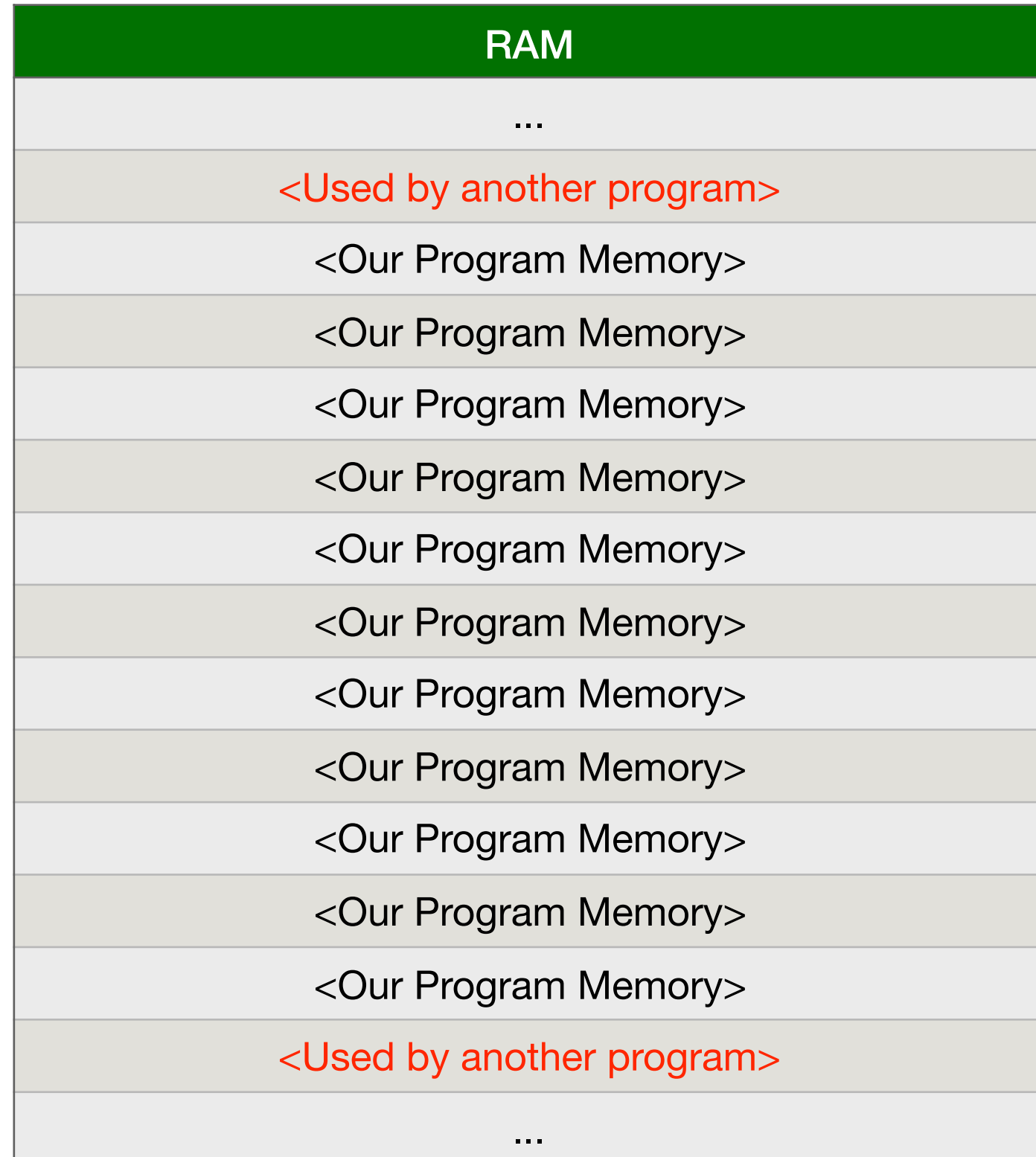
# Let's Talk About Memory

- Significantly faster than reading/writing to disk
  - Even with an SSD
- Significantly more expensive than disk space



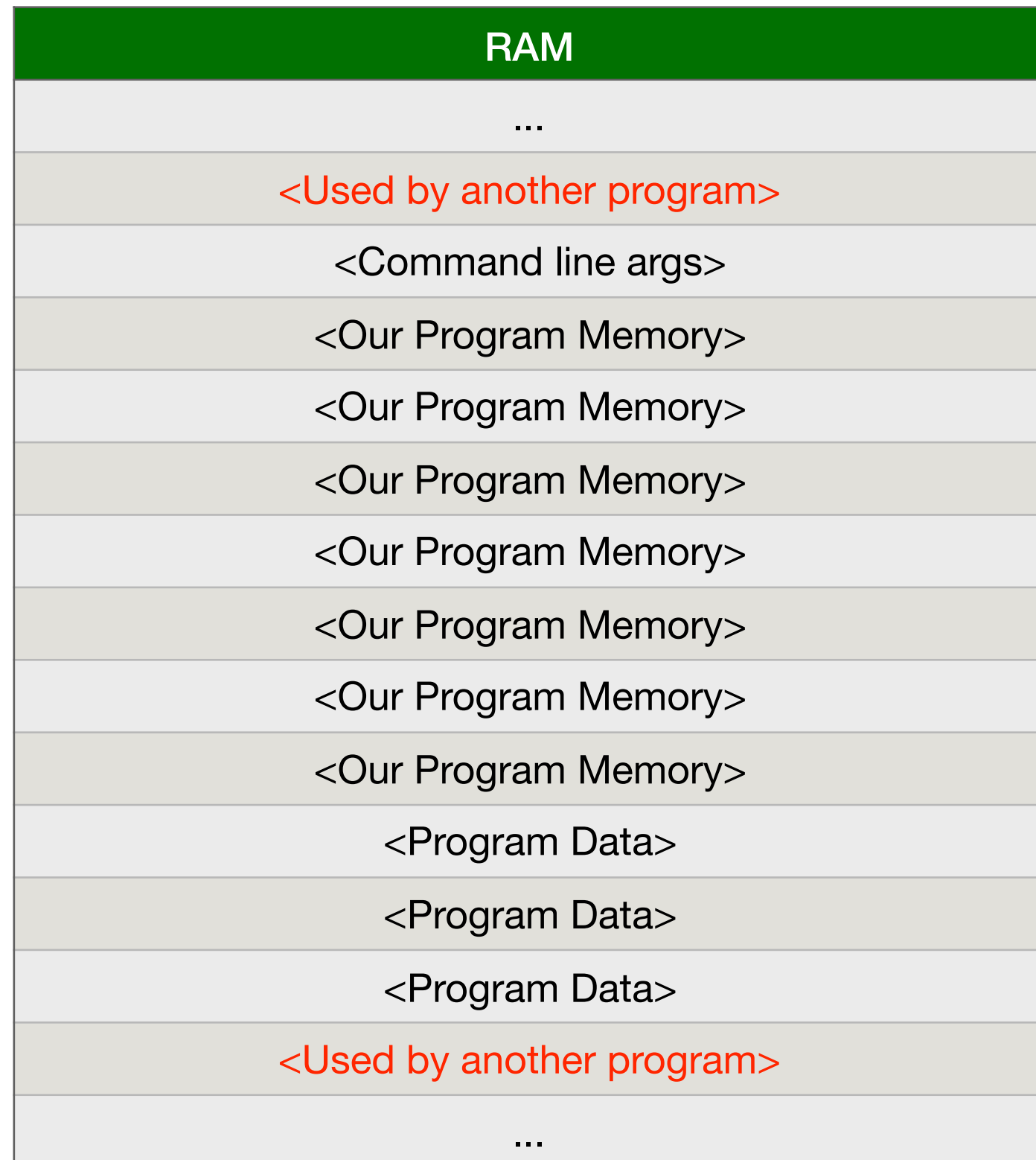
# Let's Talk About Memory

- Operating System (OS) controls memory
- On program start, OS allocates a section of memory for our program
  - Gives access to a range of memory addresses/indices



# Program Memory

- Some space is reserved for program data
- Details not important to CSE116
- The rest will be used for our data
- Data stored in the **memory stack**



# Memory Stack

- Stores the variables and values for our programs
- LIFO - Last In First Out
  - New values are added to the end of the stack
  - Only values at the end of the stack can be removed

# Memory Stack

- Method calls create new stack frames
  - Active stack frame is the currently executing method
  - Only stack values in the current stack frame can be accessed
  - A stack frame is isolated from the rest of the stack
- Program execution begins in the main method stack frame



# Memory Stack

- Code blocks control variable scope
  - Code executing within a code block (ex. if, for, while) begins a new section on the stack
- Similar to stack frames, but values outside of the code block can be accessed
- Variables/Values in the same code block cannot have the same name
  - If variable in different blocks have the same name, the program searching the inner-most code block first for that variable
- When the end of a code block is reached, all variables/values created within that block are destroyed



# Memory Stack Example

```
function computeFactorial(n){  
    result = 1  
    for (i=1; i<=n; i++) {  
        result *= i  
    }  
    return result  
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

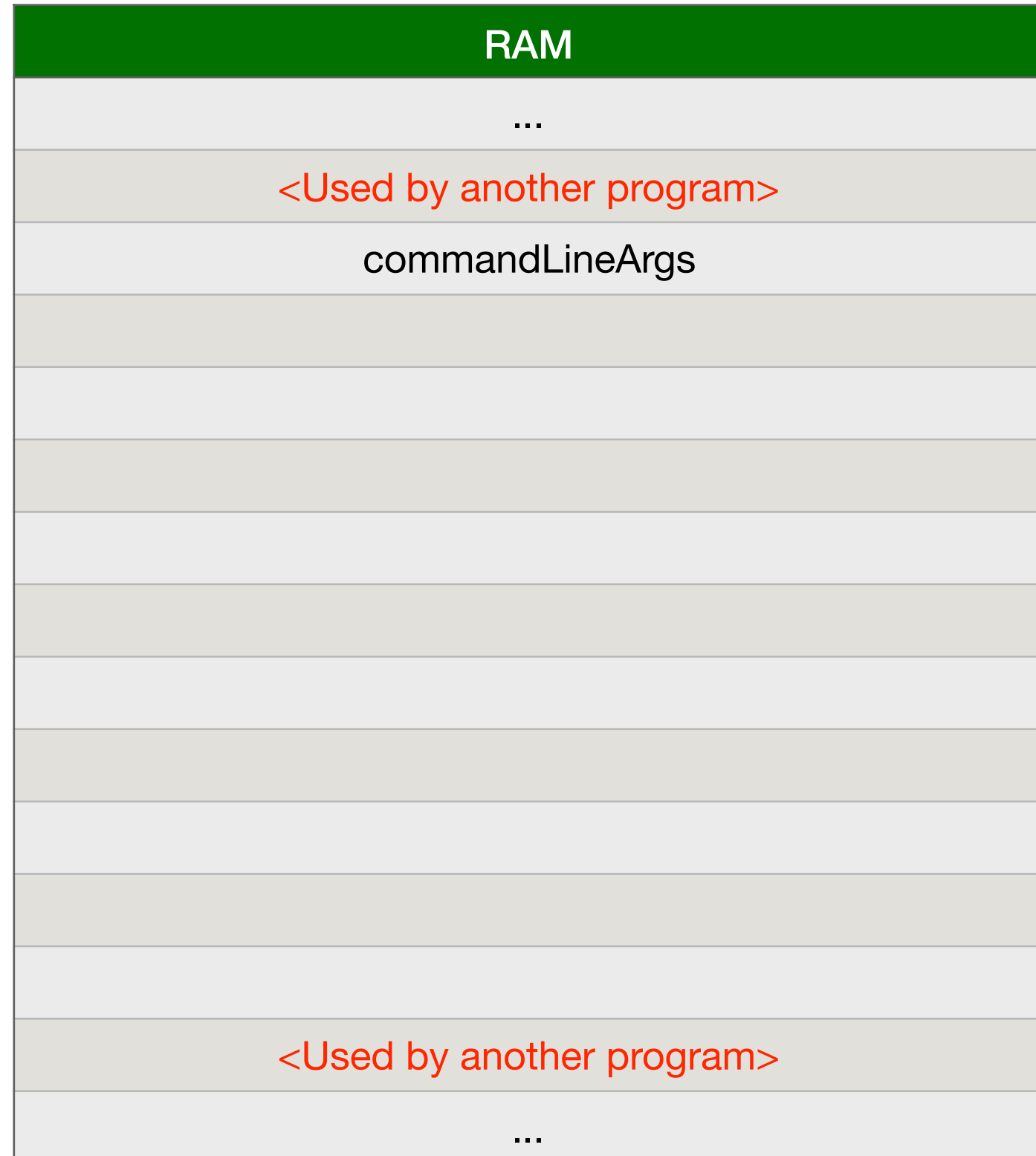
**Note: This example is language independent and will focus on the concept of memory. Each language will have differences in how memory is managed**

# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
➡ function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Command line arguments added to the stack

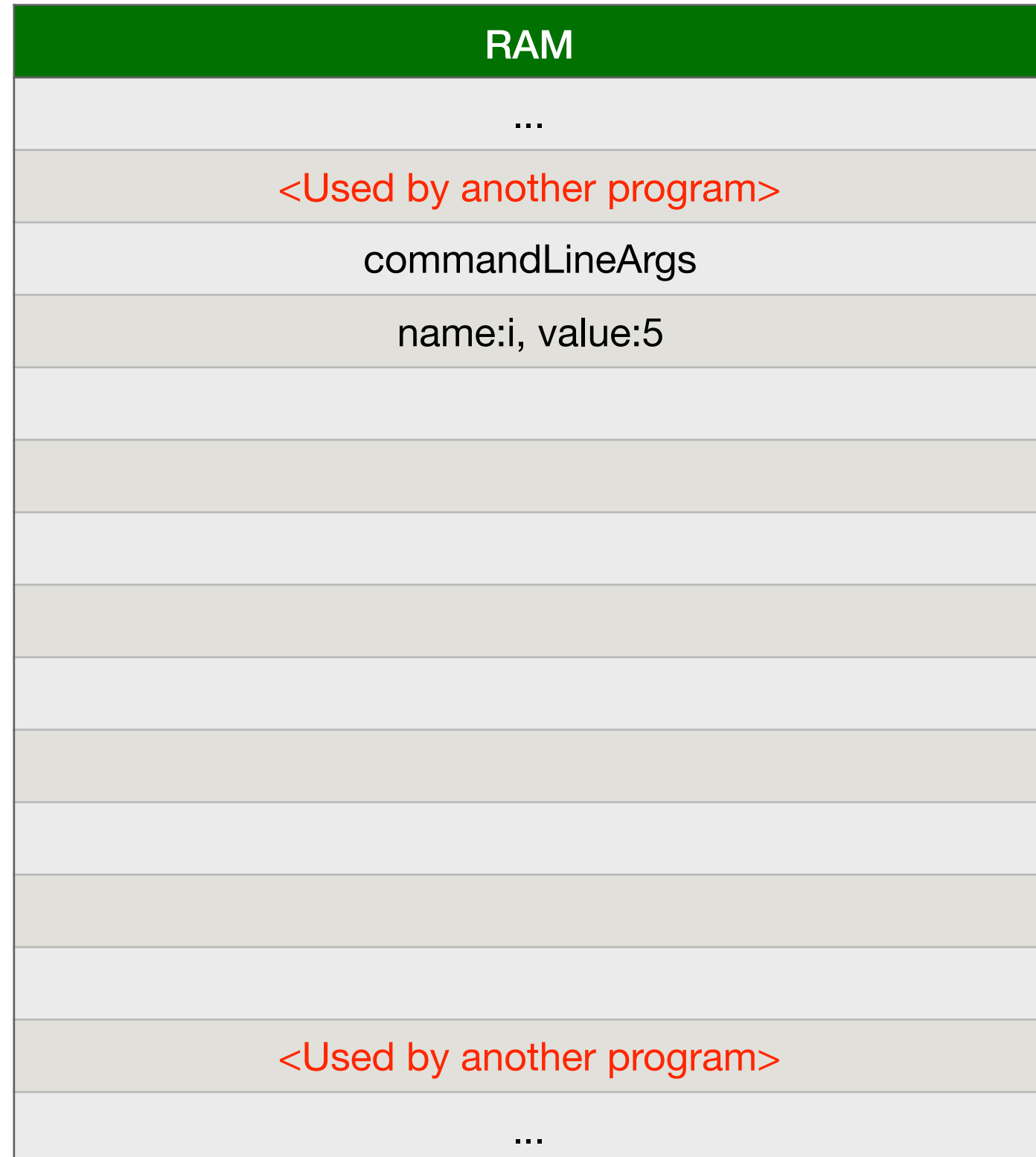


# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    → i = 5  
      n = computeFactorial(i)  
      print(n)  
}
```

- A variable named `i` of type `Int` is added to the stack
- The variable `i` is assigned a value of 5



# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    i = 5  
    → n = computeFactorial(i)  
    print(n)  
}
```

- The program enters a call to `computeFactorial`
- A new **stack frame** is created for this call





# Memory Stack Example

```
function computeFactorial(n){  
→ result = 1  
  for (i=1; i<=n; i++) {  
    result *= i  
  }  
  return result  
}  
  
function main(commandLineArgs){  
  i = 5  
  n = computeFactorial(i)  
  print(n)  
}
```

- Add result to the stack and assign it the value 1

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:1
<Used by another program>
...

# Memory Stack Example

```
function computeFactorial(n){  
    result = 1  
→ for (i=1; i<=n; i++) {  
        result *= i  
    }  
    return result  
}  
  
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Begin loop block
- Add i to the stack and assign it the value 1
  - This is different from the i declared in main since they are in different frames

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:1
<begin loop block>
name:i, value:1
<Used by another program>
...

# Memory Stack Example

```
function computeFactorial(n){  
    result = 1  
    for (i=1; i<=n; i++) {  
→    result *= i  
    }  
    return result  
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Iterate through the loop
- look for variable named result in current stack frame
  - Found it outside the loop block
  - Update it's value (remains 1 on first iteration)

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:1
<begin loop block>
name:i, value:1
<Used by another program>
...



# Memory Stack Example

```
function computeFactorial(n){  
    result = 1  
→ for (i=1; i<=n; i++) {  
        result *= i  
    }  
    return result  
}  
  
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Iterate through the loop
- look for variable named i in current stack frame
  - Found it inside the loop block
  - \*Some languages look outside the current frame

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:1
<begin loop block>
name:i, value:2
<Used by another program>
...

# Memory Stack Example


```
function computeFactorial(n){  
    result = 1  
    for (i=1; i<=n; i++) {  
→      result *= i  
    }  
    return result  
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Multiply result by i

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:2
<begin loop block>
name:i, value:2
<Used by another program>
...

# Memory Stack Example


```
function computeFactorial(n){  
    result = 1  
    for (i=1; i<=n; i++) {  
        result *= i  
    }  
     return result  
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Iterate through the loop until conditional is false

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:120
<begin loop block>
name:i, value:5
<Used by another program>
...

# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    
    return result
}

function main(commandLineArgs){
    i = 5
    n = computeFactorial(i)
    print(n)
}
```

- End of a code block is reached
- Delete ALL stack storage used by that block!
  - The variable i fell out of scope and no longer exists

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
name:n, value:5
name:result, value:120
<Used by another program>
...

# Memory Stack Example



```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- End of a function is reached
- Delete ALL stack storage used by that stack frame!
- Replace function call with its return value

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
function returned: 120
<Used by another program>
...

# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    i = 5  
    → n = computeFactorial(i)  
      print(n)  
}
```

- Declare n
- Assign return value to n

[illegible]

# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- Print n to the screen
- At this point:
  - No memory of variables n (function), i (function), or result

[illegible]

# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- End of program
- Free memory back to the OS



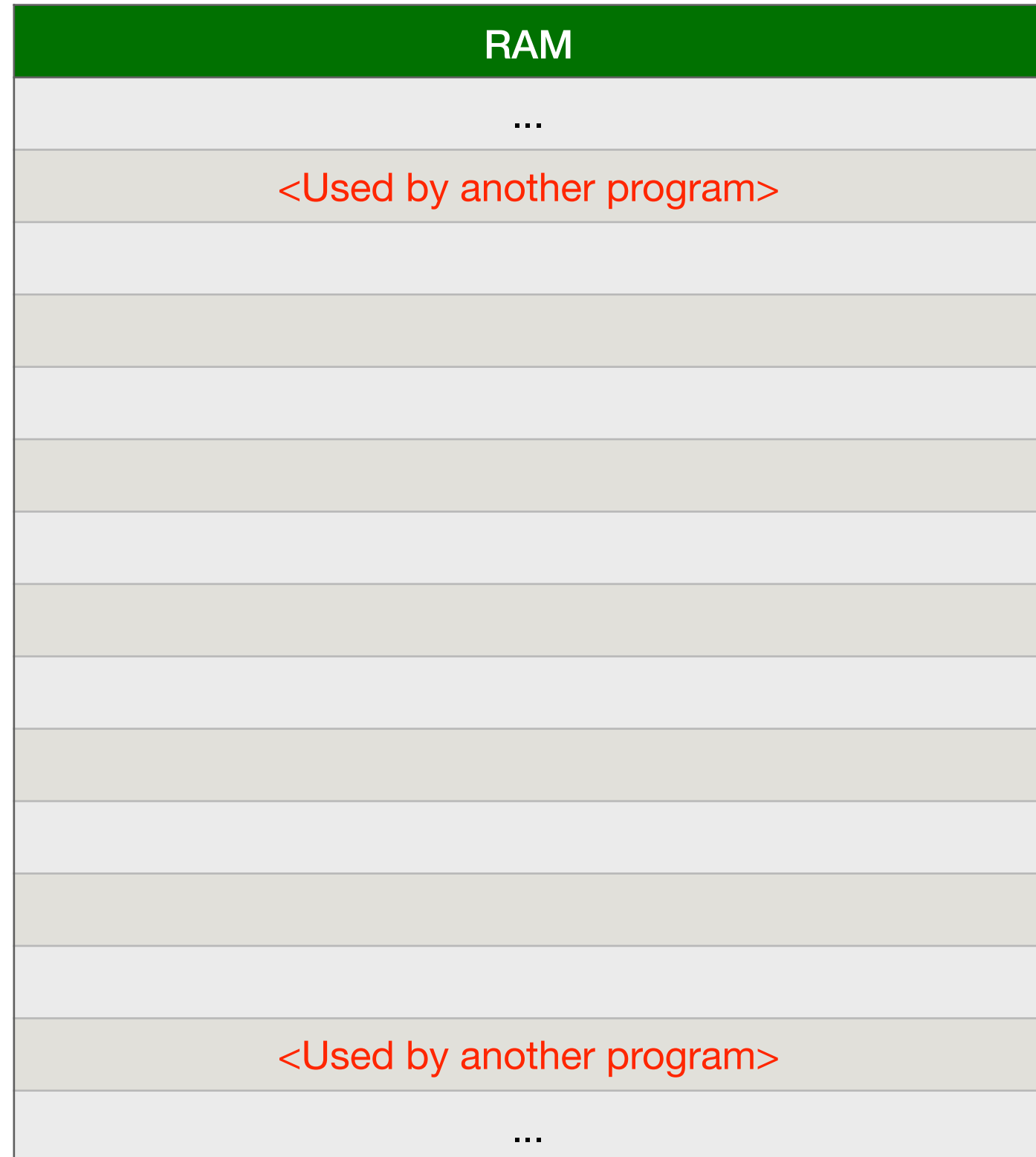


# Memory Stack Example

```
function computeFactorial(n){
    result = 1
    for (i=1; i<=n; i++) {
        result *= i
    }
    return result
}
```

```
function main(commandLineArgs){  
    i = 5  
    n = computeFactorial(i)  
    print(n)  
}
```

- No memory of our program



# Stack Memory

- Only "primitive" types are stored in stack memory
  - Double/Float
  - Int/Long/Short
  - Char
  - Byte
  - Boolean
- Values corresponding to Java primitives
  - Compiler converts these objects to primitives

# Stack Memory

- Only "primitive" types are stored in stack memory
  - 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

What if our data needs to change size?

# Memory Heap

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5 (main)
name:data, value>List of Ints (main)
name:n, value:120 (main)
<Used by another program>
...

```
...
data.addValue(78)
...
```

- Variable data has values before and after it in memory
- Where do we store 78?
  - On the heap

# Memory Heap

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5 (main)
name:data, value:97197 (main)
name:n, value:120 (main)
<Used by another program>
...

RAM @97197
...
List of Ints
List of Ints
...

- Heap memory is dynamic
- Can be anywhere in RAM
  - Location not important
  - Location can change
- Use references to find data
  - Variable data only stores a reference to the List of Ints

# Memory Heap

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5 (main)
name:data, value:97197 (main)
name:n, value:120 (main)
<Used by another program>
...

RAM @97197
...
List of Ints
List of Ints
...

- Objects *usually* stored in heap memory
- [In Scala] Int, Double, Boolean, Char, and few others are all stored on the stack
- All other types are stored in the heap, including every type you define

# Memory Heap Example

RAM
...
<Used by another program>
commandLineArgs
name:data, value:38772 (main)
<Used by another program>
...

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

RAM @38772
...
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:38772 (main)
<function call stack frame>
name:input, value:38772 (function)
<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 @38772
...
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:38772 (main)
<function call stack frame>
name:input, value:38772 (function)
<Used by another program>
...

- Add 1 to input.state variable
- Find the object at memory address 38772
- Alter the state of the object at that address



RAM @38772
...
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:38772 (main)
<Used by another program>
...

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



RAM @38772
...
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:38772 (main)
<Used by another program>
...

- Access data.stateVar
- Find the object at memory address 38772
- Access the state of the object at that address

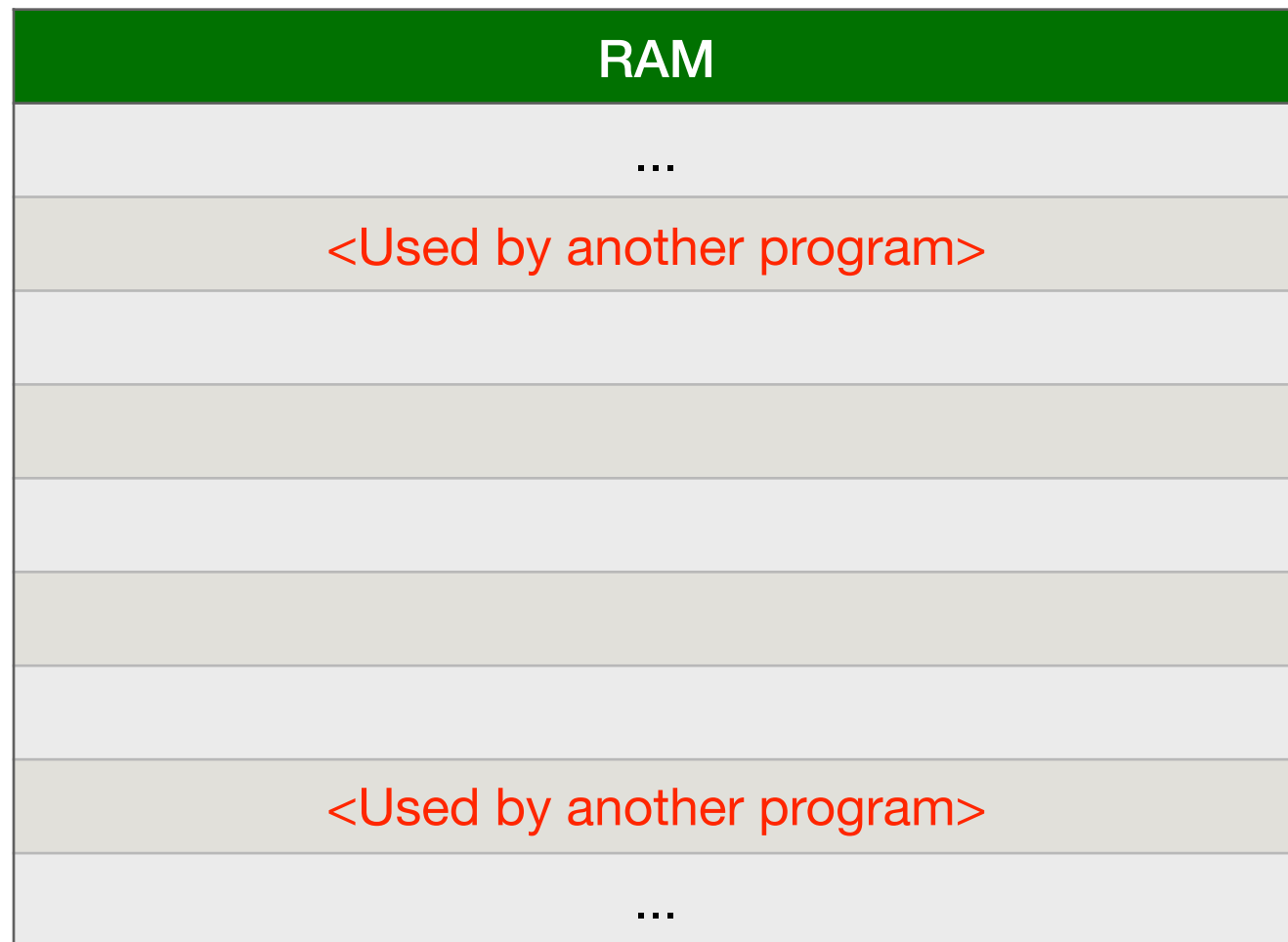


RAM @38772
...
ClassWithStateObject
-stateVar value:1
...

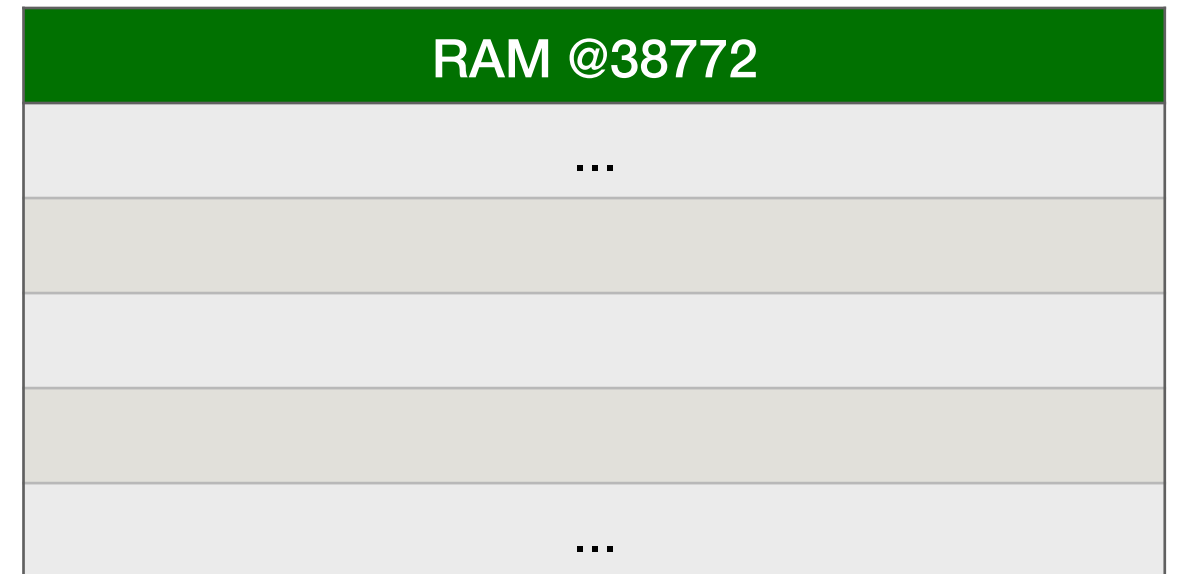
```
class ClassWithState{  
    int stateVar = 0;  
}
```

```
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 Question

**Question:** In a package named "execution" create a Scala **class** named "Team" and a Scala **object** named "Referee".

Team will have:

- State values of type Int representing the strength of the team's offense and defense with a constructor to set these values. The parameters for the constructor should be offense then defense
- A third state variable named "score" of type Int that is not in the constructor, is declared as a **var**, and is initialized to 0

Referee will have:

- A method named "playGame" that takes two Team objects as parameters and return type Unit. This method will alter the state of each input Team by setting their scores equal to their offense minus the other Team's defense. If a Team's offense is less than the other Team's defense their score should be 0 (no negative scores)
- A method named "declareWinner" that takes two Teams as parameters and returns the Team with the higher score. If both Teams have the same score, return a new Team object with offense and defense both set to 0

# Lecture Question

## Sample Usage

```
val t1: Team = new Team(7, 3)
val t2: Team = new Team(4, 20)
```

```
Referee.playGame(t1, t2)
assert(Referee.declareWinner(t1, t2) == t2)
assert(Referee.declareWinner(t2, t1) == t2)
```

## Commentary

We create Team as a **class** since we want to create many objects of type Team that will compete against each other. Each team will have different state (offense, defense, score), but will be the same type (Team)

Referee is an **object** since there only needs to be one of them and the object has no state. The same referee can officiate every game between any two teams

We pass **references** of objects of type Team to the Referee. Since the Referee has the references, when it changes the score of a Team that change is made to the state of that Team throughout the program