Model of Execution

Lecture Task

- This is Lecture Task 5 from the Pale Blue Dot project -

- In the pbd.PaleBlueDot object, write a method named "greaterCircleDistance" which:

  - Takes two Lists of Doubles representing the latitude and longitude of two locations on Earth:

    - you may assume that each list contains 2 elements which are the latitude and longitude in that order

  - Returns the greater circle distance between the two input points in kilometers as a Double

    - Use 6371 km as the radius of Earth

- This site (https://www.movable-type.co.uk/scripts/latlong.html) provides JavaScript code that computes greater circle distance. Your task for this objective is to convert this code to a Scala method. You are not expected to understand all the trigonometry

- In tests.LectureTask5, complete a test suite to test the functionality of this method

  - Use the link above to generate test cases

Interpretation v. Compilation

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

```python
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

```python
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

```scala
class CompilerError(var state: Int) {

  def addToState(toAdd: Int): Unit ={
    println("adding to state")
    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

```scala
class CompilerError(var state: Int) {

  def addToState(toAdd: Int): Unit ={
    println("adding to state")
    this.state += toAdd
  }

}

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

Compilation - Scala

- Scala compiles to Java Byte Code

- Executed by the Java Virtual Machine (JVM)

 - Installed on Billions of devices!
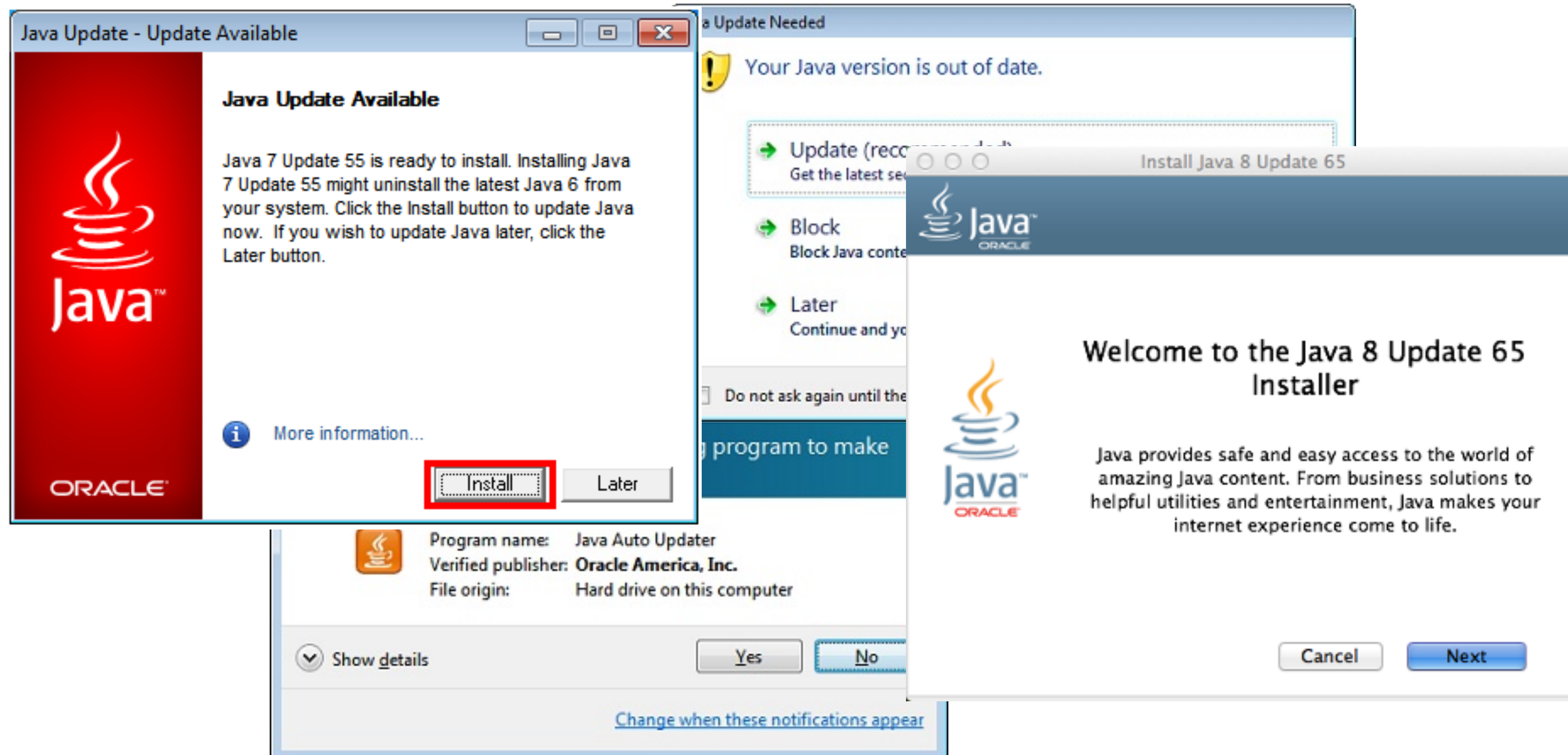
Compilation - Scala

- 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

  - Math in Scala is Java's Math class

  - We'll sometimes use Java libraries in this course
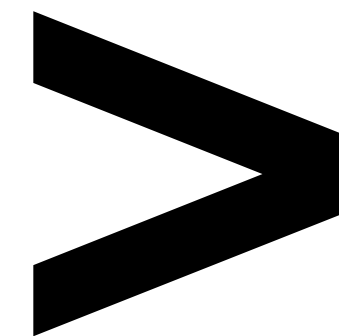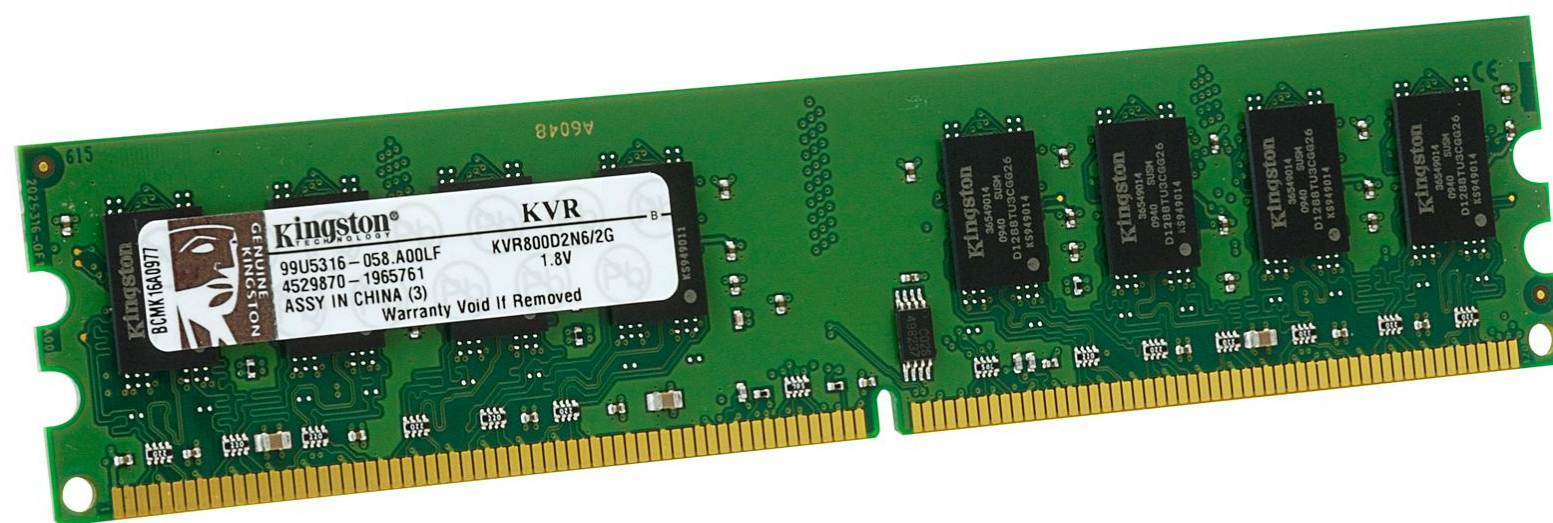
Memory

Let's Talk About Memory

- Random Access Memory (RAM)

  - Access any value by index

  - Effectively a giant array

- All values in your program are stored here
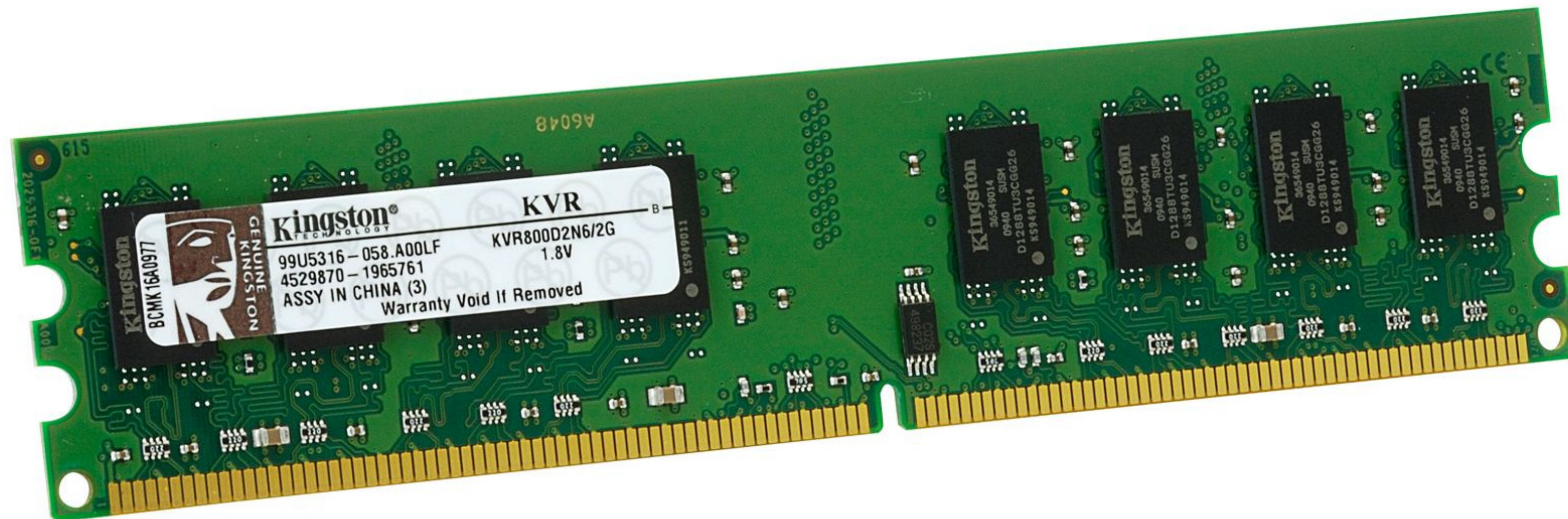
Let's Talk About Memory

- 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

RAM
...
<Used by another program>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Used by another program>
...
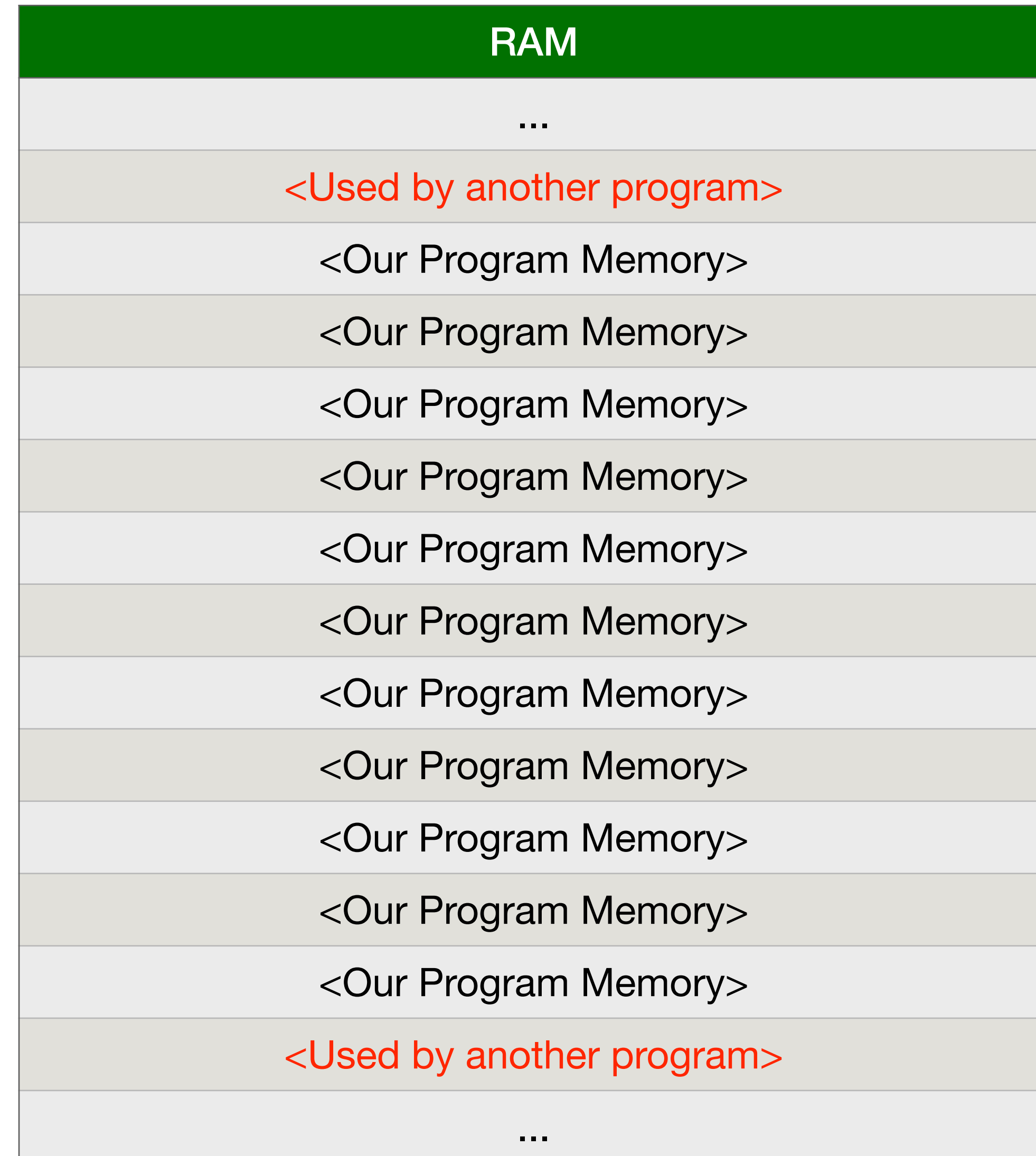
Program Memory

- 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**

RAM
...
<Used by another program>
<Command line args>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Our Program Memory>
<Program Data>
<Program Data>
<Program Data>
<Used by another program>
...
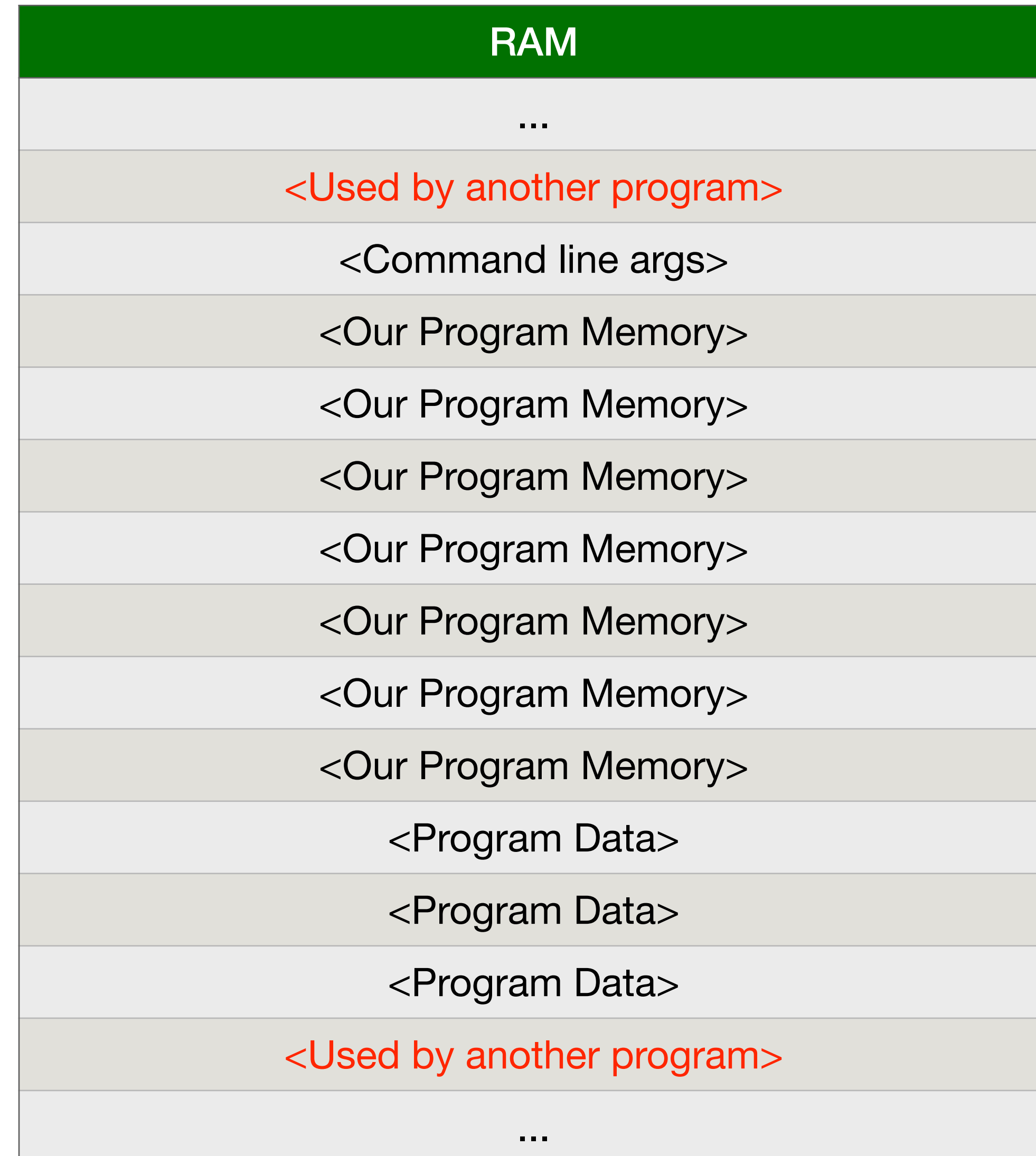
Memory Stack

- 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

Variable Scope

- 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

Variable Scope

- Variables/Values in the same code block cannot have the same name

  - If variables in different blocks have the same name, the program searches 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** (Removed from 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)
}
```
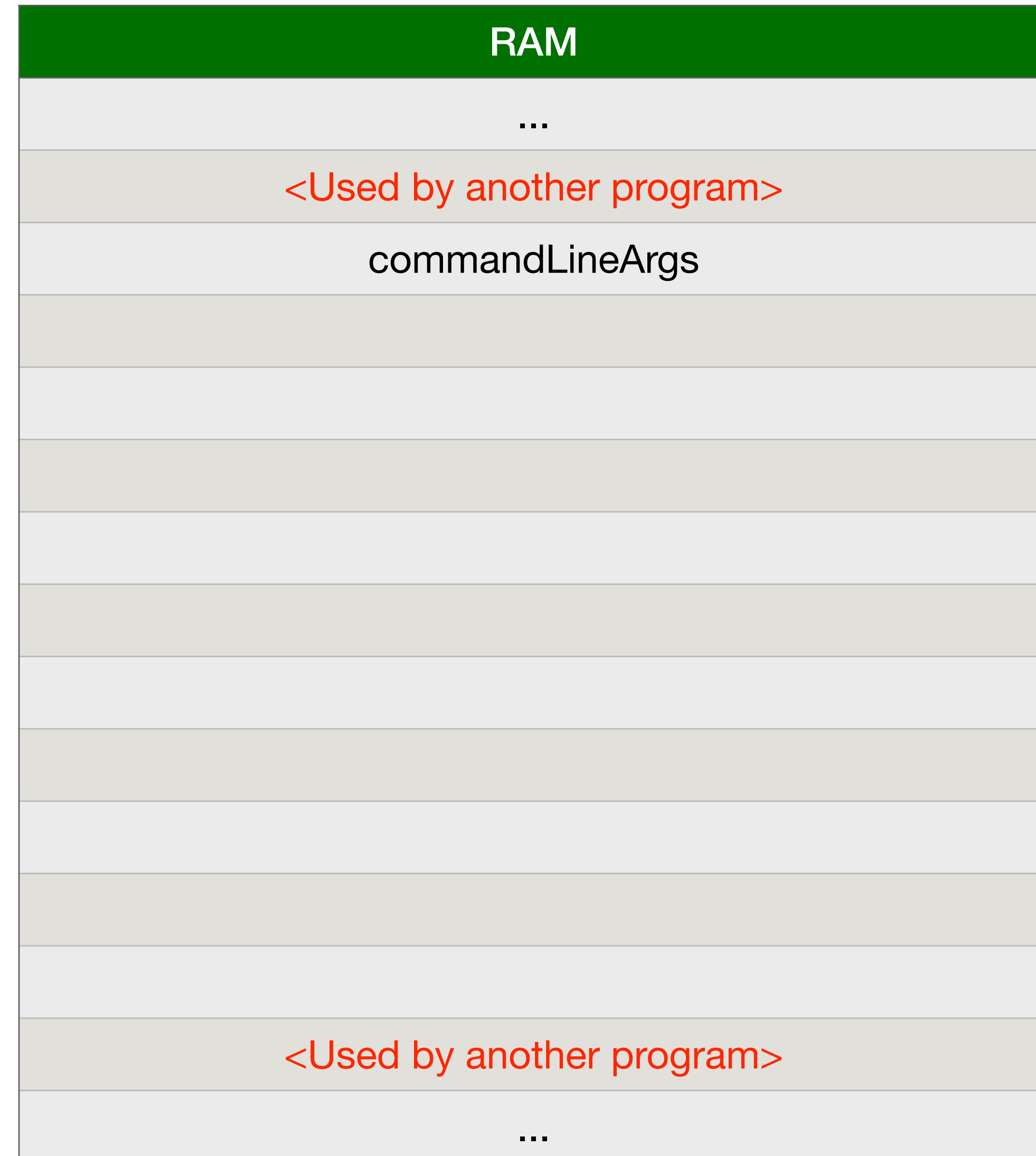
Note: This example is language independent and will focus on the concept of memory. Each language will have differences in how memory is managed

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

RAM
...
<Used by another program>
commandLineArgs
<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)
}
```

- A variable named i of type Int is added to the stack

- The variable i is assigned a value of 5

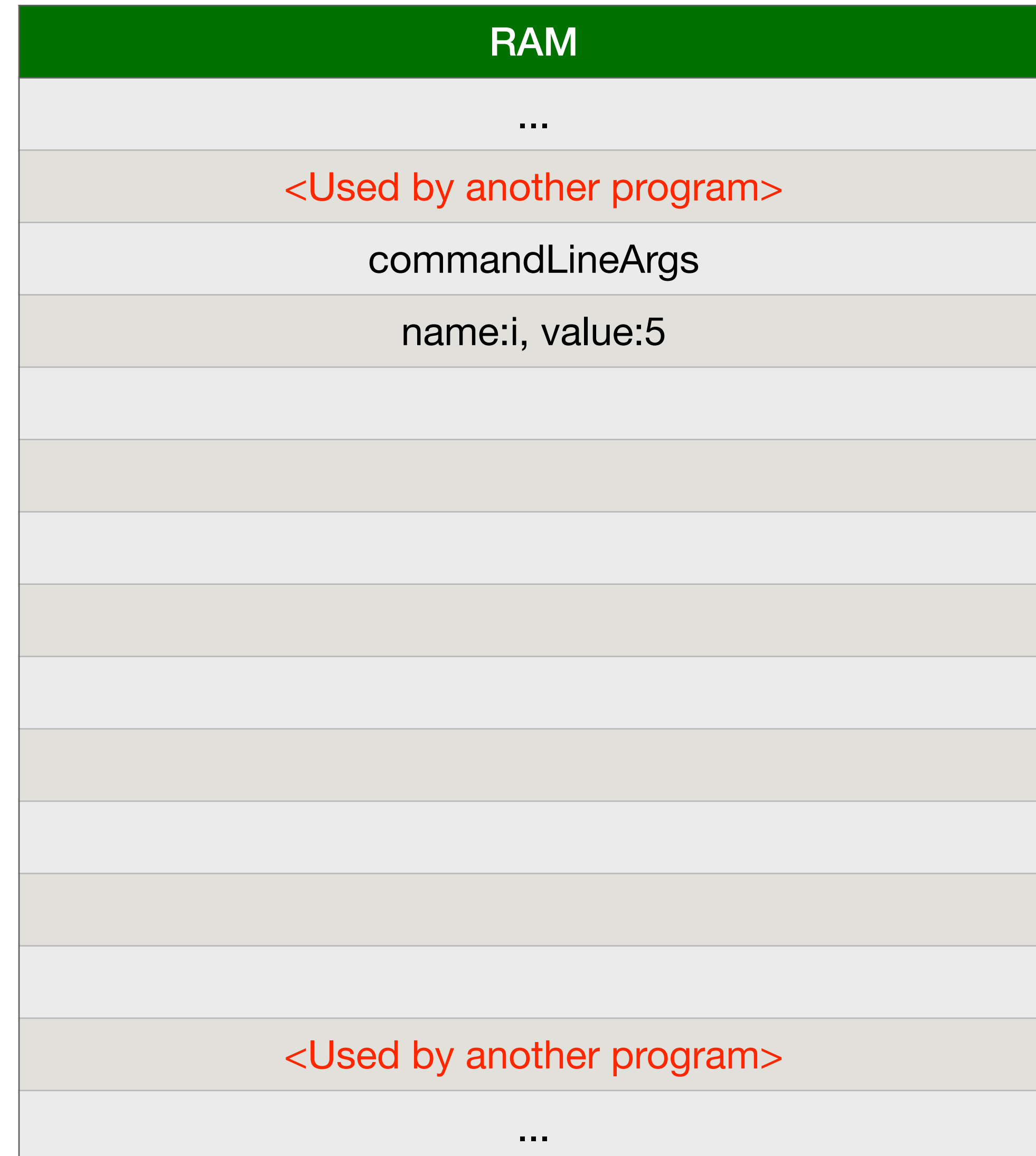| RAM |
|---|
| ... |
| <Used by another program> |
| commandLineArgs |
| name:i, value:5 |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| <Used by another program> |
| ... |

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)
}
```

- The program enters a call to computeFactorial

- A new **stack frame** is created for this call

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
<begin computeFactorial stack frame>
<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)
}
```

- Add n to the stack and assign it the value from the input argument

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

- 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)
}
```
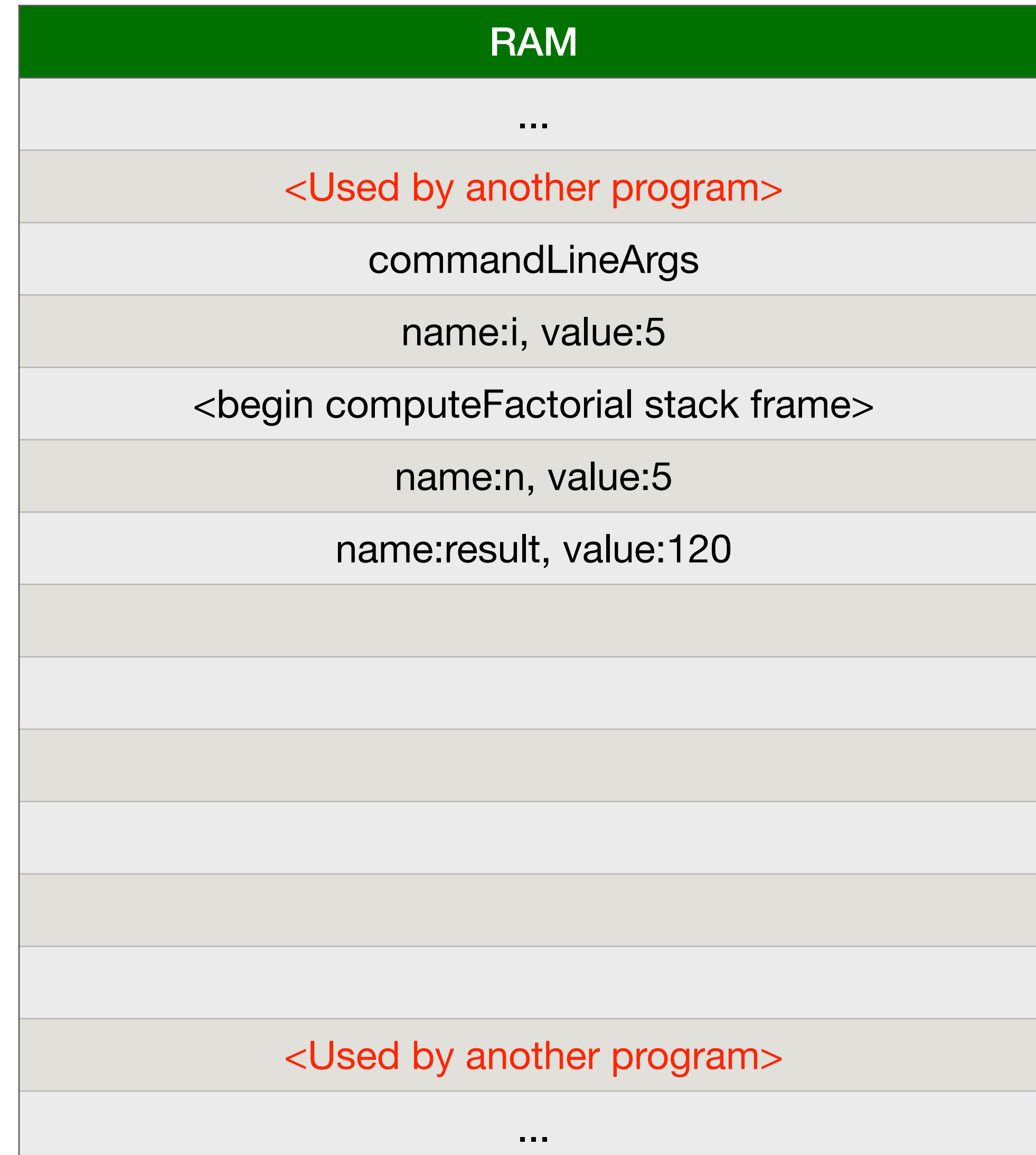
- Iterate through the loop until conditional is false

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 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 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

RAM
...
<Used by another program>
commandLineArgs
name:i, value:5
name:n, value: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)
}
```

- Print n to the screen

- At this point:

  - No memory of variables n (function), i (function), or result

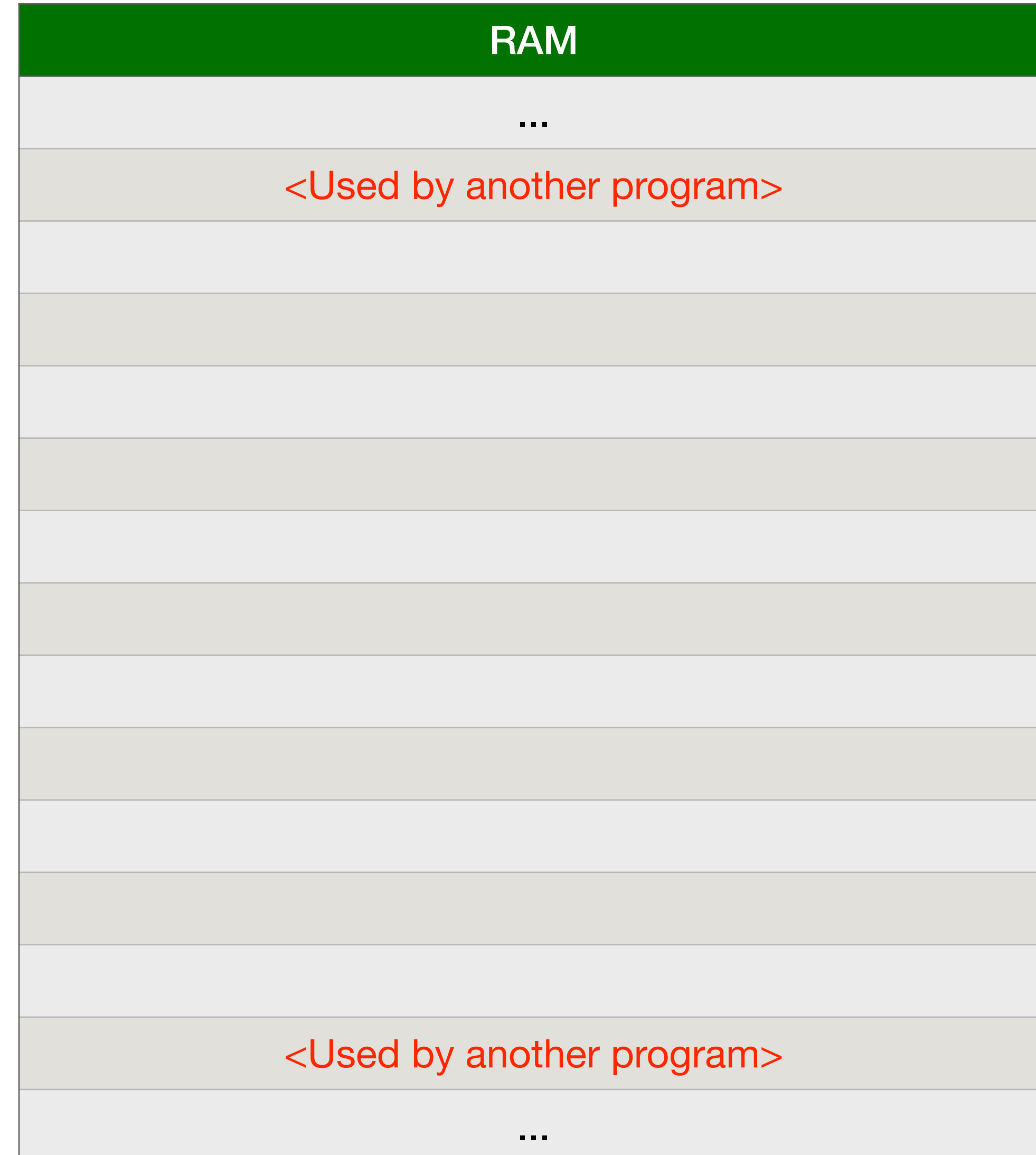| RAM |
| --- |
| ... |
| <Used by another program> |
| commandLineArgs |
| name:i, value:5 |
| name:n, value:120 |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| <Used by another program> |
| ... |

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)
}
```

- End of program

- Free memory back to the OS

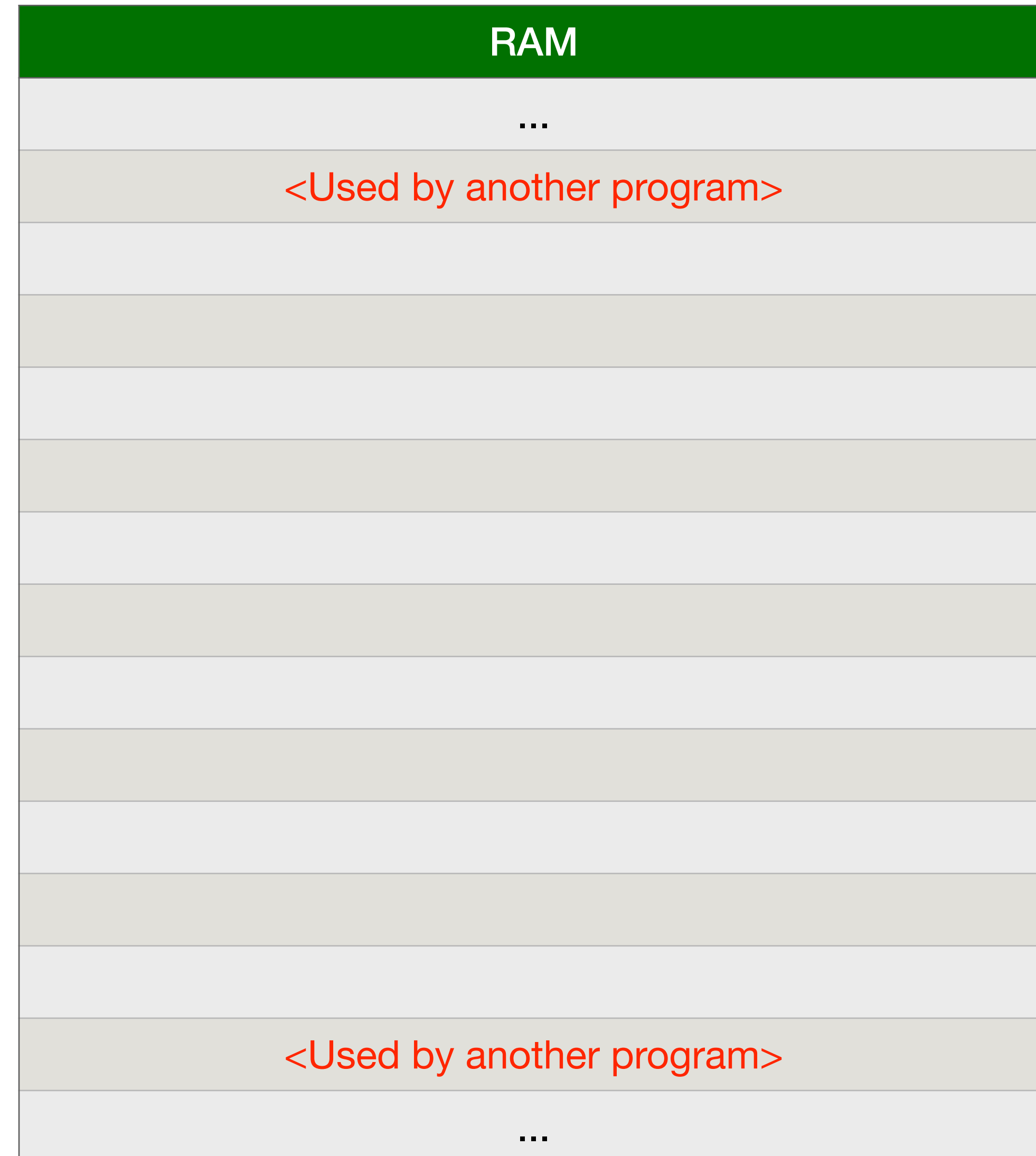| RAM |
|-----|
| ... |
| <Used by another program> |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| <Used by another program> |
| ... |

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)
}
```
→

- No memory of our program

😢

RAM
…
<Used by another program>
<Used by another program>
…

Stack Memory

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

  - Double/Float

  - Int/Long/Short

  - Char

  - Byte

  - Boolean

- Values corresponding to Java primitives

  - Compiler converts these objects to primitives in Java Byte Code

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

Lecture Task

- This is Lecture Task 5 from the Pale Blue Dot project -

- In the pbd.PaleBlueDot object, write a method named "greaterCircleDistance" which:

  - Takes two Lists of Doubles representing the latitude and longitude of two locations on Earth:

    - you may assume that each list contains 2 elements which are the latitude and longitude in that order

  - Returns the greater circle distance between the two input points in kilometers as a Double

    - Use 6371 km as the radius of Earth

- This site (https://www.movable-type.co.uk/scripts/latlong.html) provides JavaScript code that computes greater circle distance. Your task for this objective is to convert this code to a Scala method. You are not expected to understand all the trigonometry

- In tests.LectureTask5, complete a test suite to test the functionality of this method

  - Use the link above to generate test cases