Model of Execution

With Memory Diagrams

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

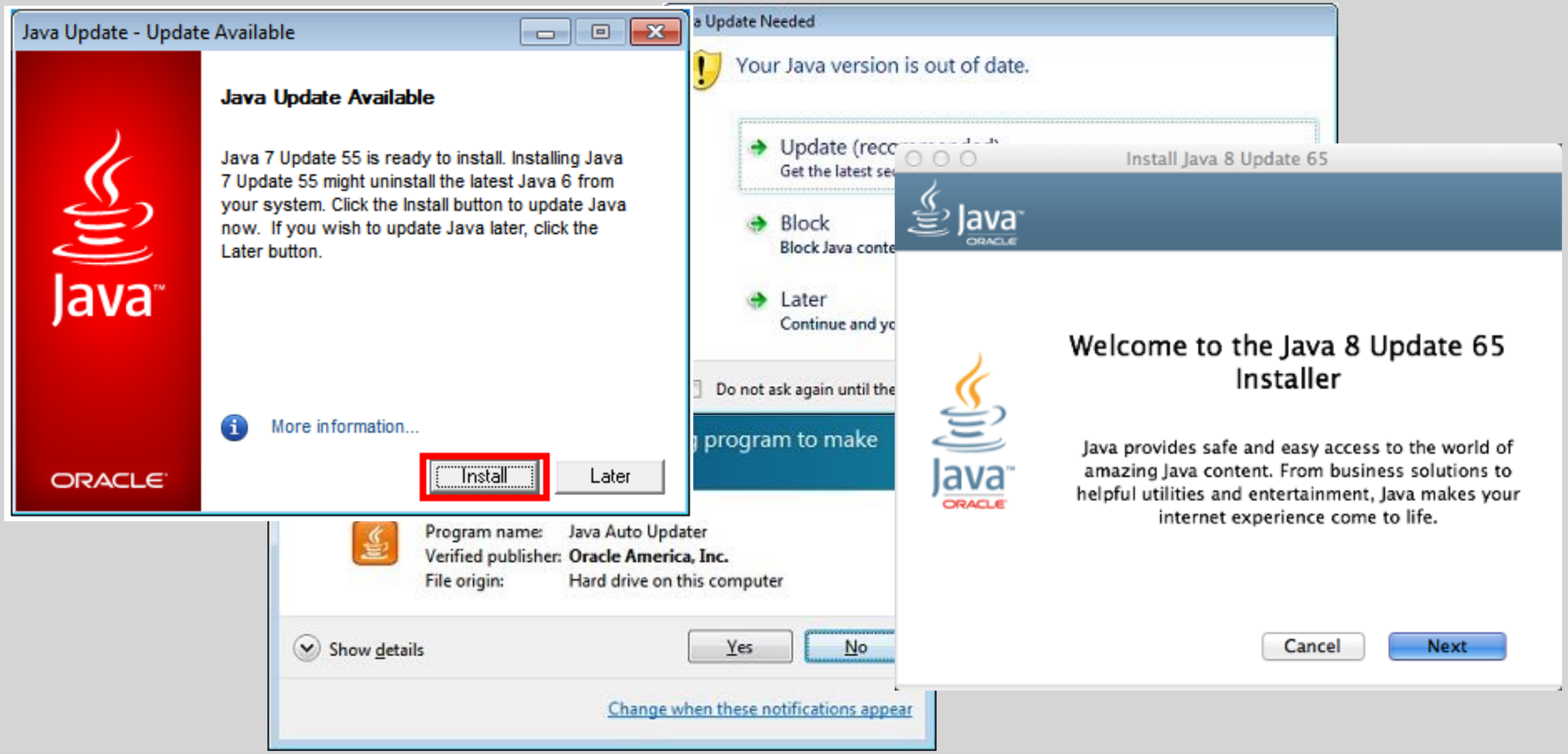
```
class CompilerError(var state: Int) {

  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

```
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!



Java Update - Update Available

Java Update Available

Java 7 Update 55 is ready to install. Installing Java 7 Update 55 might uninstall the latest Java 6 from your system. Click the Install button to update Java now. If you wish to update Java later, click the Later button.

a Update Needed

Your Java version is out of date.

program to make

Program name: Java Auto Updater
Verified publisher: Oracle America, Inc.
File origin: Hard drive on this computer

Install Java 8 Update 65

Welcome to the Java 8 Update 65 Installer

Java provides safe and easy access to the world of amazing Java content. From business solutions to helpful utilities and entertainment, Java makes your internet experience come to life.

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

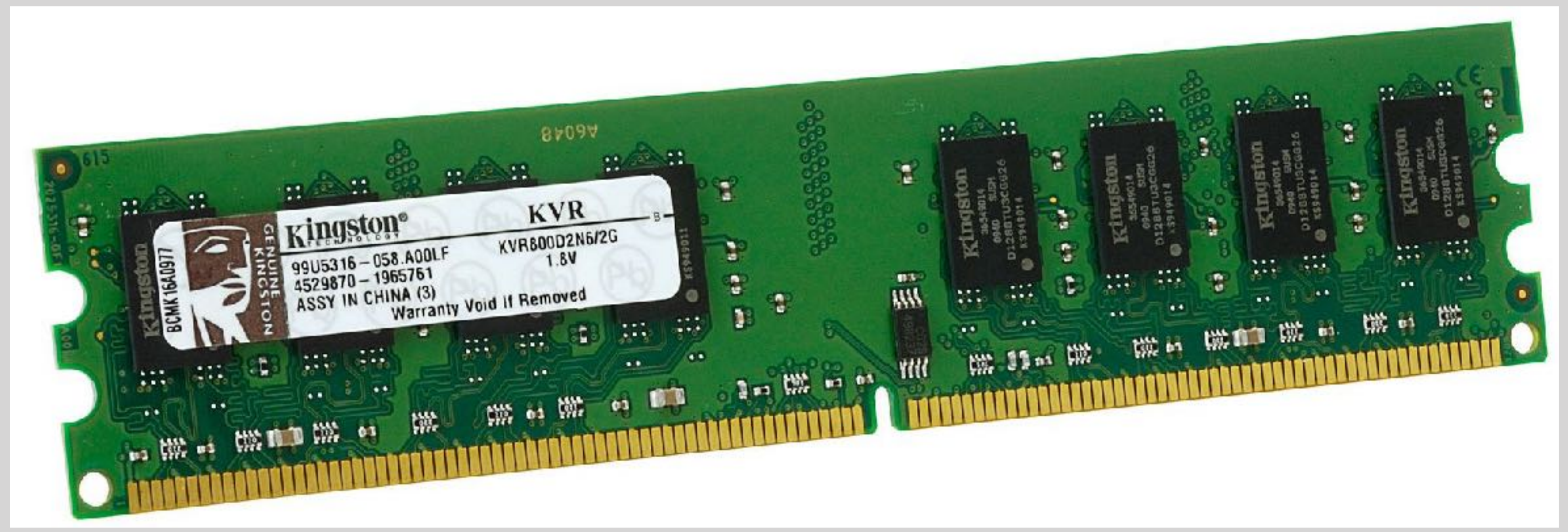
Memory

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>
<Our Program Memory>
<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

Stack Frames

- Every method call creates a new **stack frame**
  - Active stack frame is the currently executing method
  - Only variables declared 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's stack frame

Stack Frames

- In our memory diagrams:
  - Stack frames are designated by **solid** boxes

Code Blocks and 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 current code block can be accessed

Code Blocks and 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)

Code Blocks and Variable Scope

- In our memory diagrams:
  - Code blocks are designated by **dashed** boxes

Stack Memory

- Only "primitive" types are stored directly in stack memory
  - Double/Float
  - Int/Long/Short
  - Char
  - Byte
  - Boolean
  - String*

- All other objects are stored in heap memory**
  - Store reference to these object on the stack

***Strings are actually more complex, but we will treat them as though they are on the stack in this course**

****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**

Stack Memory

- Only "primitive" types are stored directly in stack memory
  - Double/Float
  - Int/Long/Short
  - Char
  - Byte
  - Boolean
  - String*

- All other objects are stored in heap memory**
  - Store reference to these object on the stack

***Strings are actually more complex, but we will treat them as though they are on the stack in this course**

****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**

Our First Memory Diagram

```scala
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

- We'll create a memory diagram of this program

Our First Memory Diagram

```scala
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

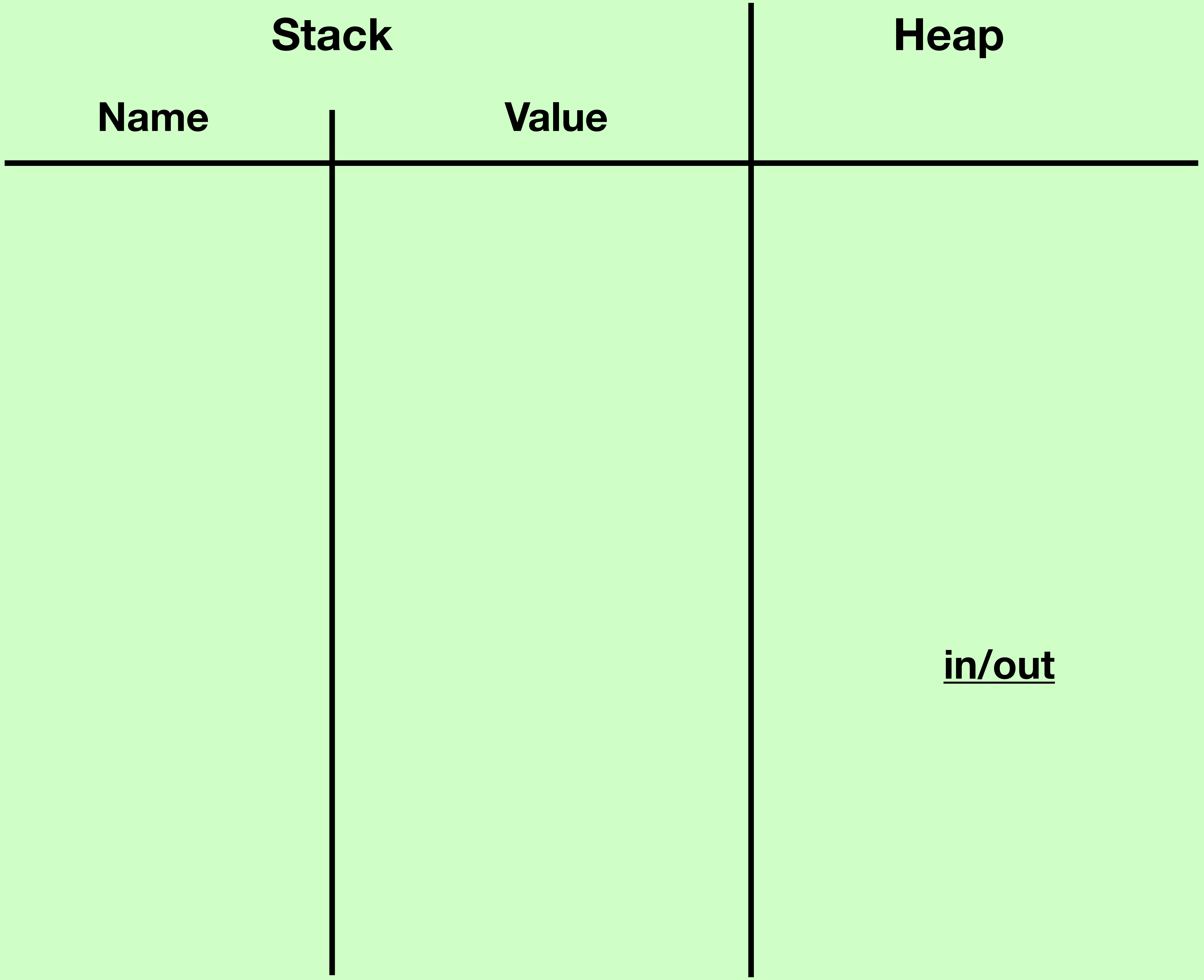
- Start by setting up the diagram
- Separate columns for stack and heap memory

Stack		Heap
Name	Value	

Our First Memory Diagram

```scala
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

- Add a section for inputs and outputs
  - This is where you write what's printed to the screen



Our First Memory Diagram

```scala
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

- Start execution at the main method

Stack		Heap
Name	**Value**	
		<u>**in/out**</u>

Our First Memory Diagram

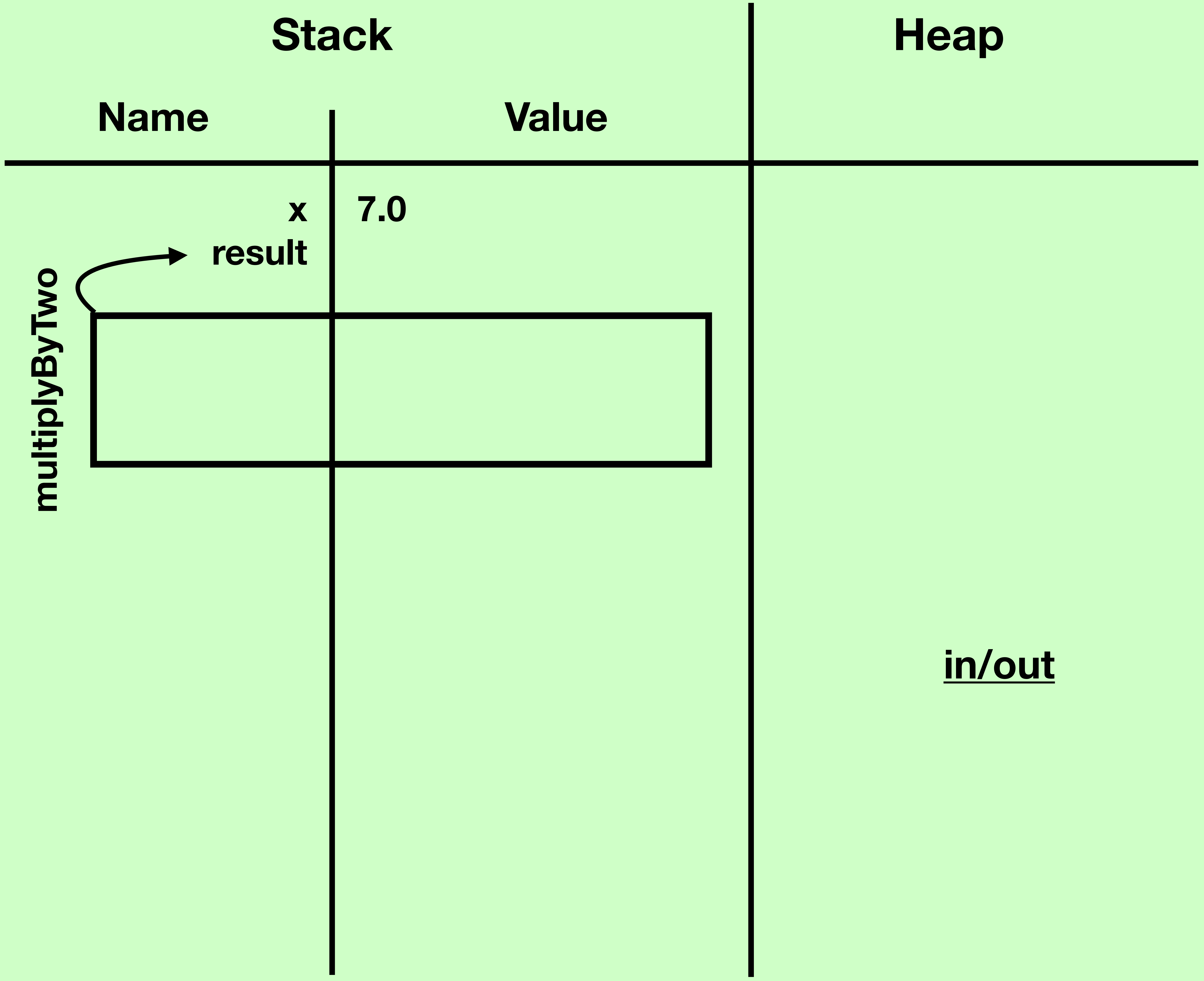
```scala
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

Stack		Heap
Name	**Value**	
x	**7.0**	
		in/out

- We have our first variable declaration
- Add the name and value to the stack

Our First Memory Diagram

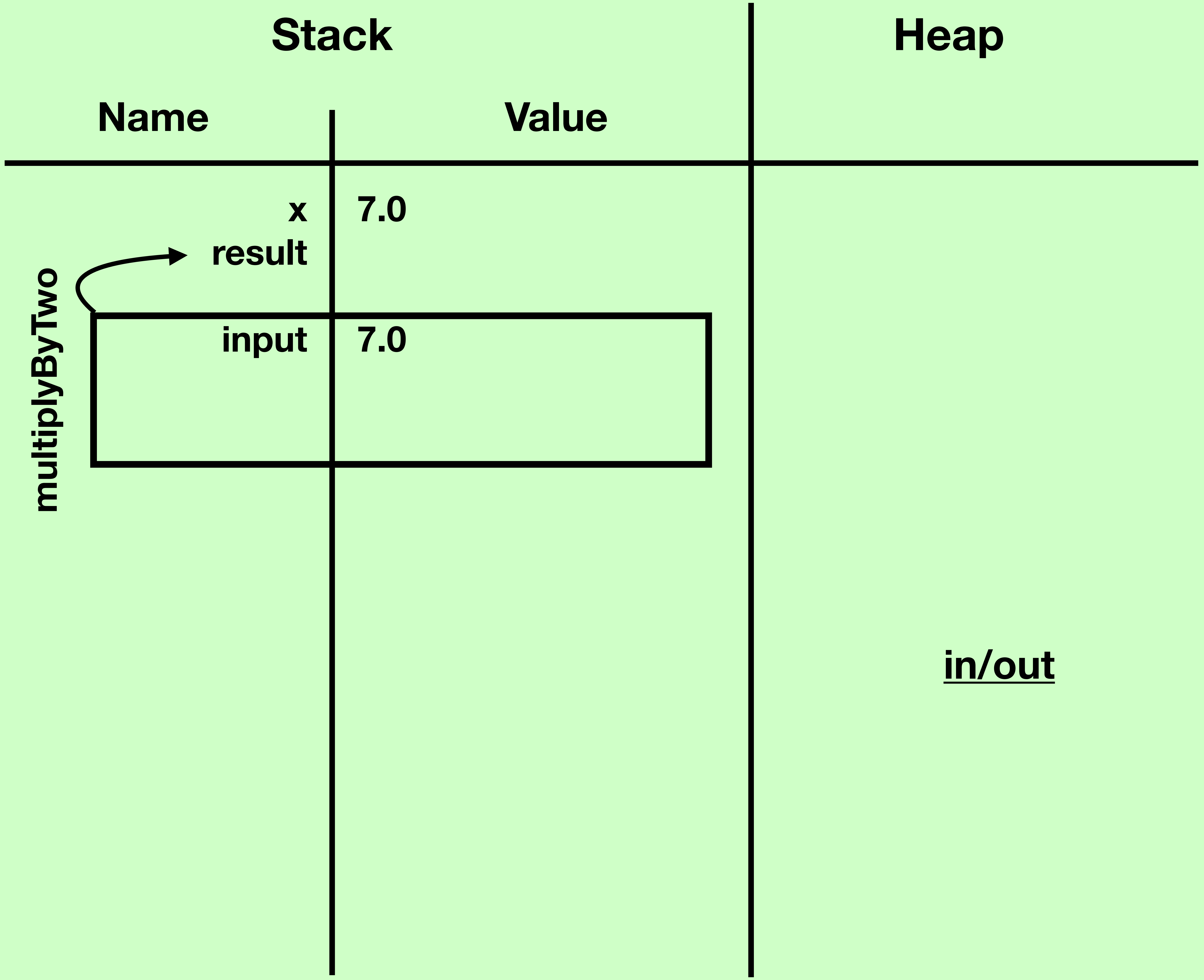
```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```



- We now have a method call
- Method calls create new stack frames

Our First Memory Diagram

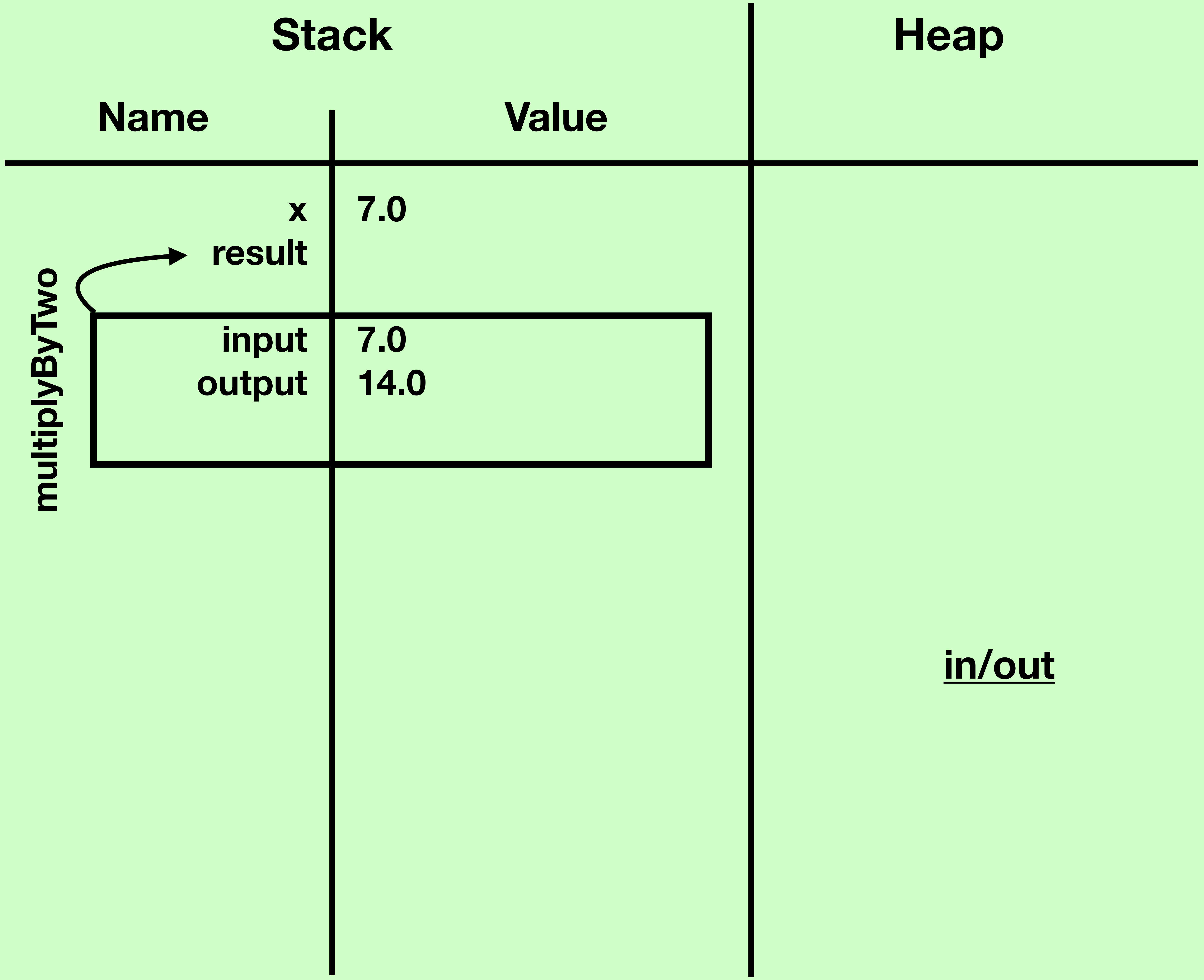
```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```



- Add the parameter to the stack inside the new frame

Our First Memory Diagram

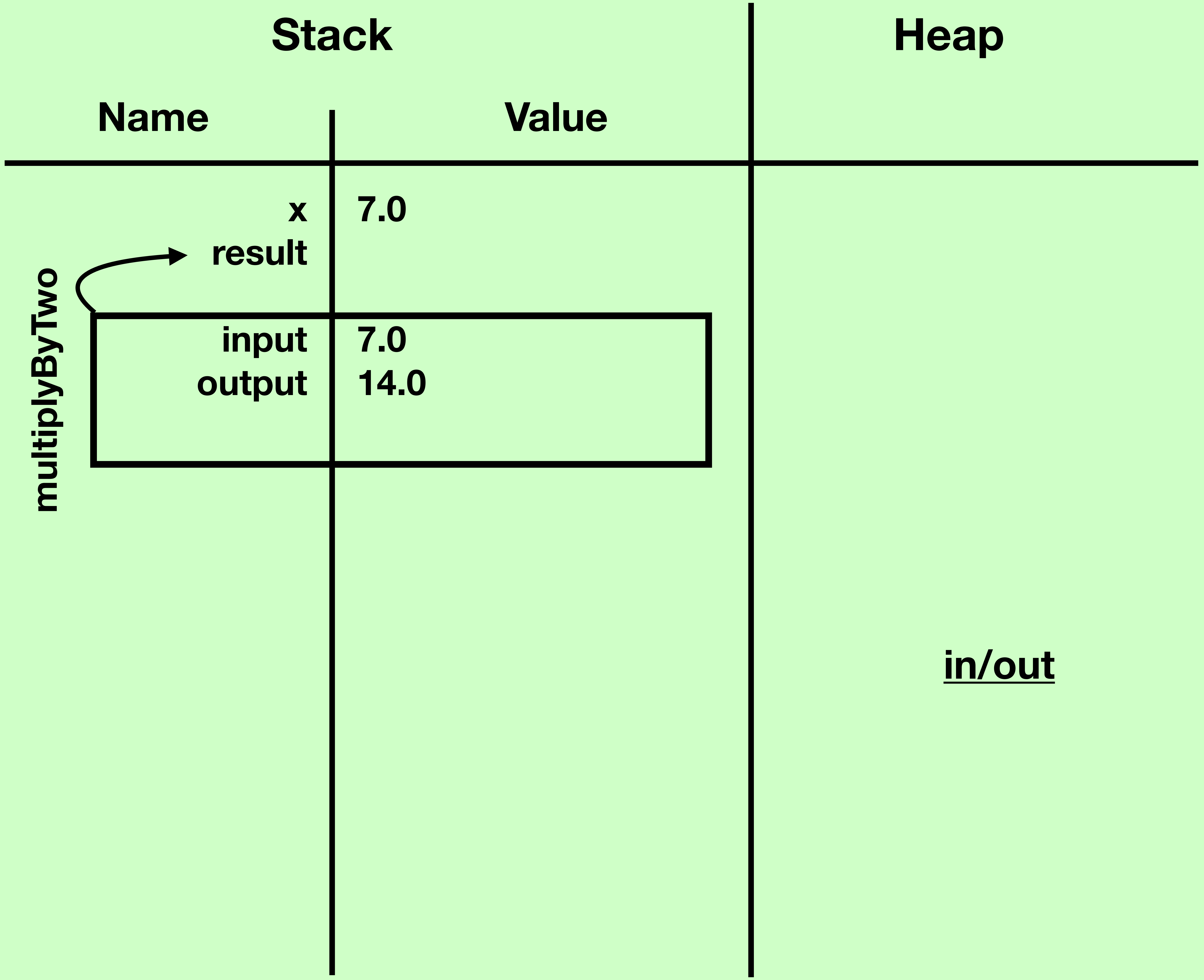
```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```



- Add new variable inside the stack frame

Our First Memory Diagram

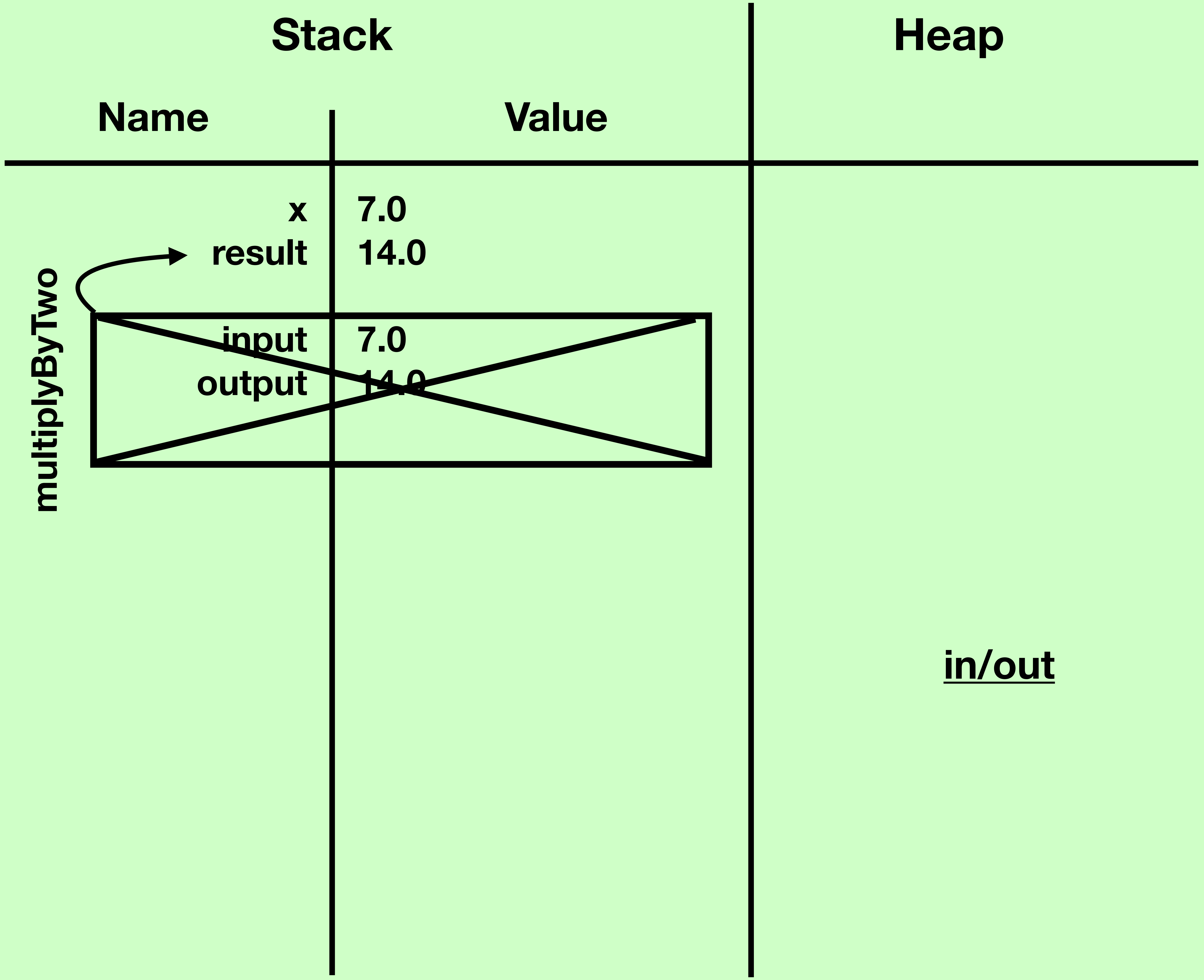
```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```



- We reach a return value
- The stack frame ends

Our First Memory Diagram

```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

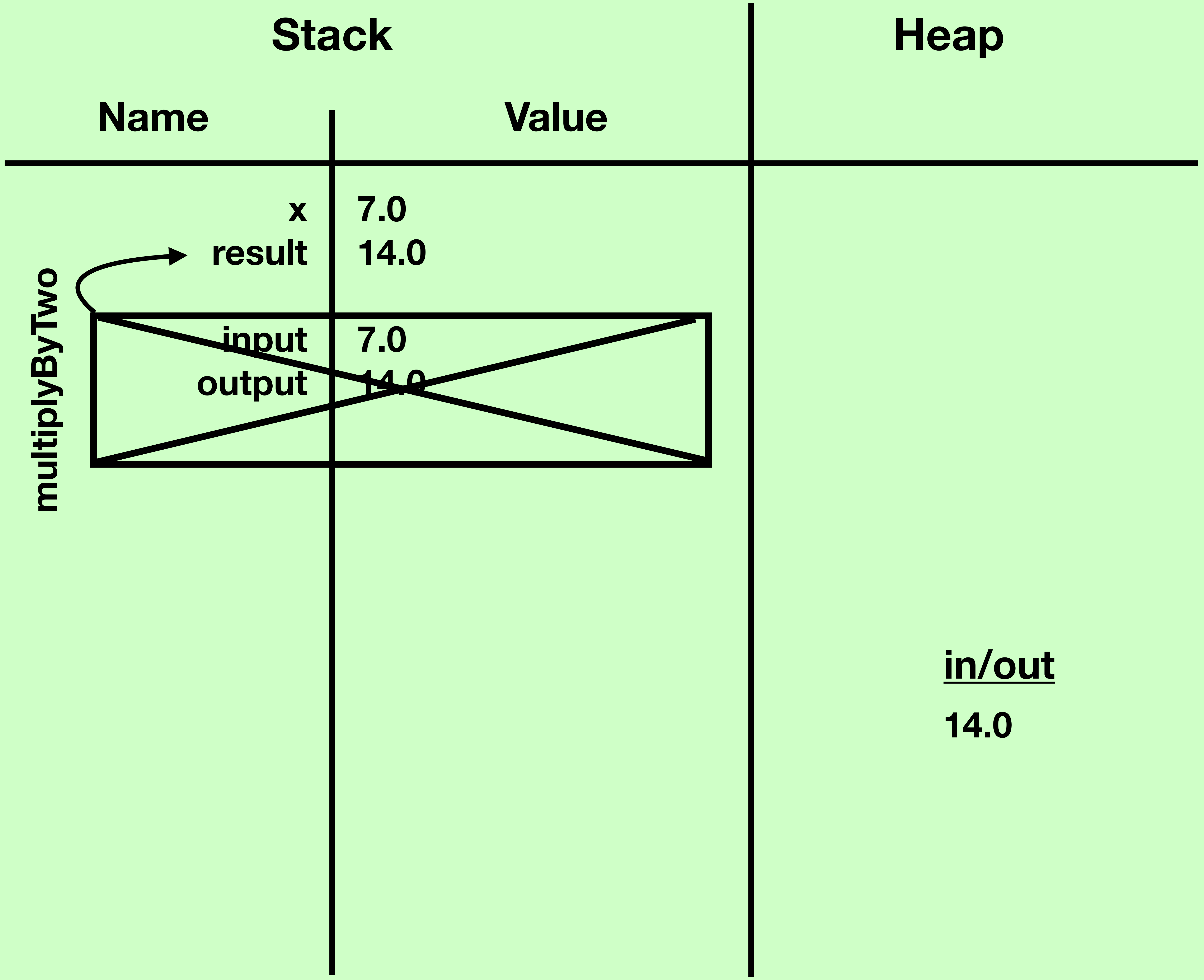


- We cross out the stack frame
- It is removed from the stack

Our First Memory Diagram

```
def multiplyByTwo(input: Double): Double = {
  val output = input * 2.0
  output
}

def main(args: Array[String]): Unit = {
  val x: Double = 7.0
  val result = multiplyByTwo(x)
  println(result)
}
```

- Print result to the screen



Stack
Heap
Name
Value
x
7.0
result
14.0
multiplyByTwo
input
7.0
output
14.0
in/out
14.0