Heap Memory

Stack Memory

- Only "primitive" values are stored directly on the stack
  - Double/Float
  - Int/Long/Short
  - Char
  - Byte
  - Boolean
  - *String*

- All other objects are stored in heap memory

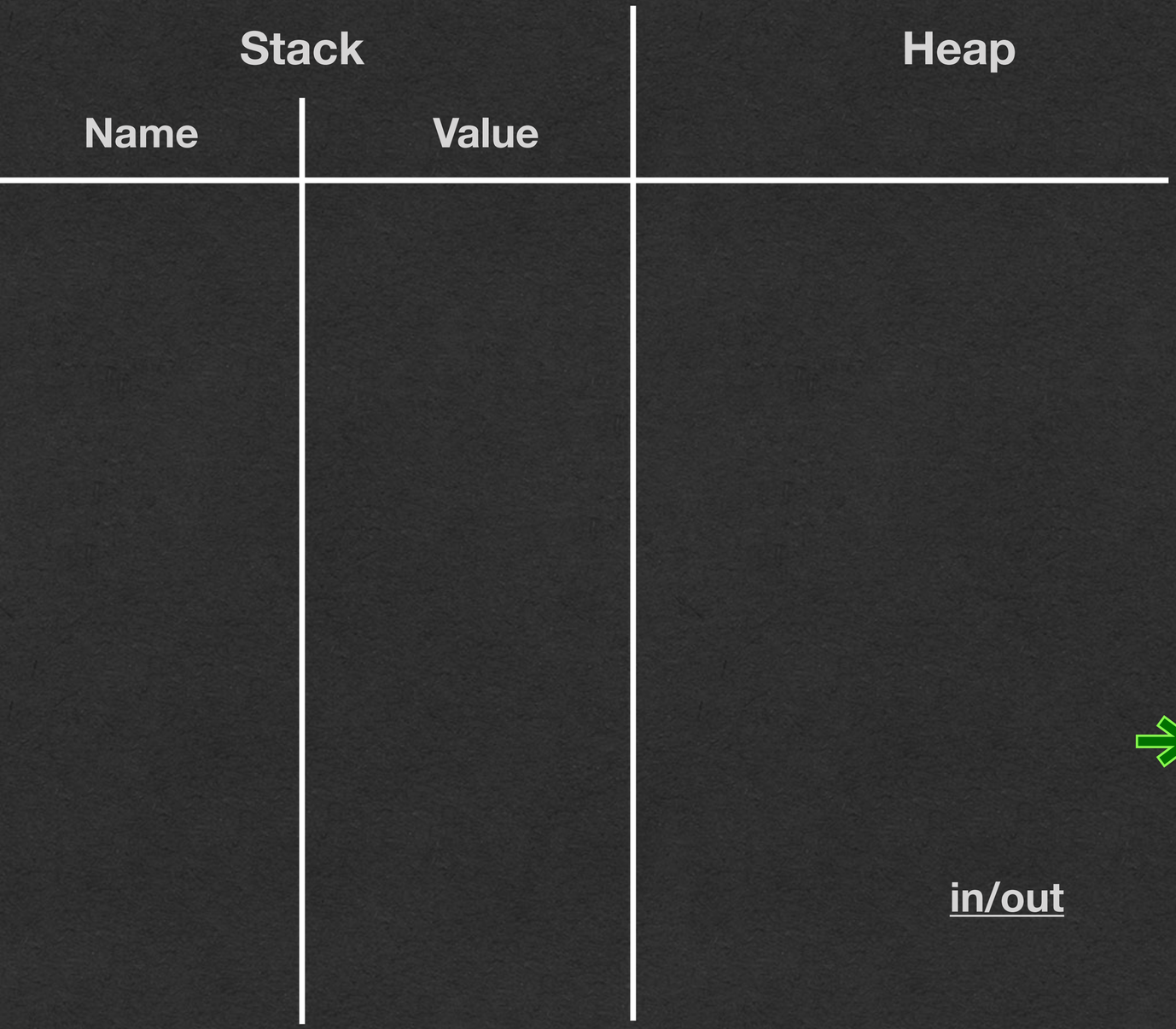
Memory Heap

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
}
```

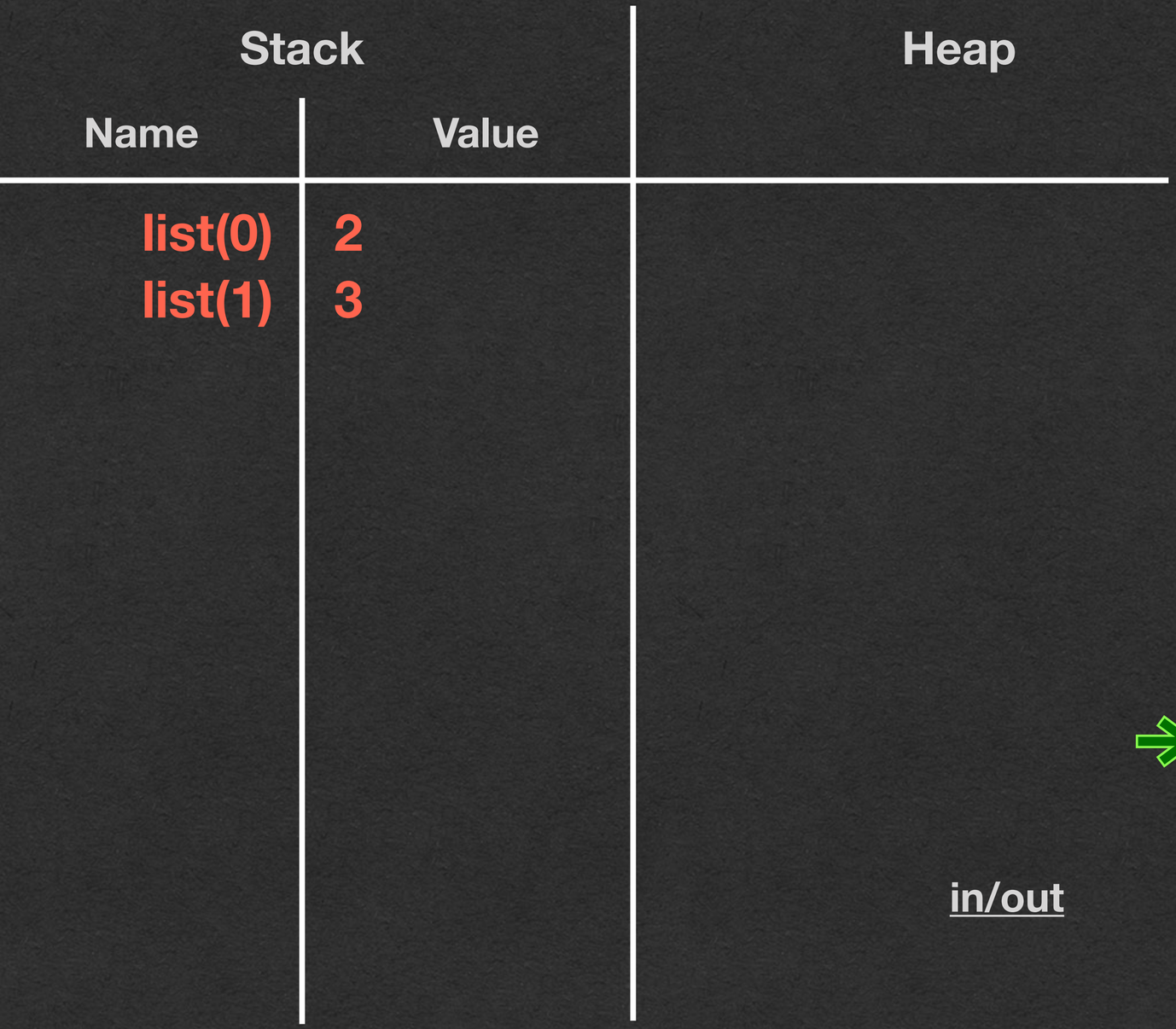
Bad Memory Heap Example



- First, let's try storing a List on the stack
  - Hint: It won't work!
  - Never do this on your memory diagrams!!

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

Bad Memory Heap Example



- 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
}
```

in/out

Bad Memory Heap Example

Stack		Heap
Name	Value	
list(0)	**2**	
list(1)	**3**	
x	5	
y	12	

- 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
}
```

in/out

Bad Memory Heap Example

Stack		Heap
Name	Value	
list(0)	**2**	
list(1)	**3**	
x	5	
y	12	

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

in/out

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

Bad Memory Heap Example

Stack		Heap
Name	Value	
list(0)	1	
list(1)	2	
list(2) x	5 3	
y	12	

in/out

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

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**

Bad Memory Heap Example

Stack		Heap
Name	Value	
x	5	
y	12	
list(0)	**1**	
list(1)	**2**	
list(2)	**3**	

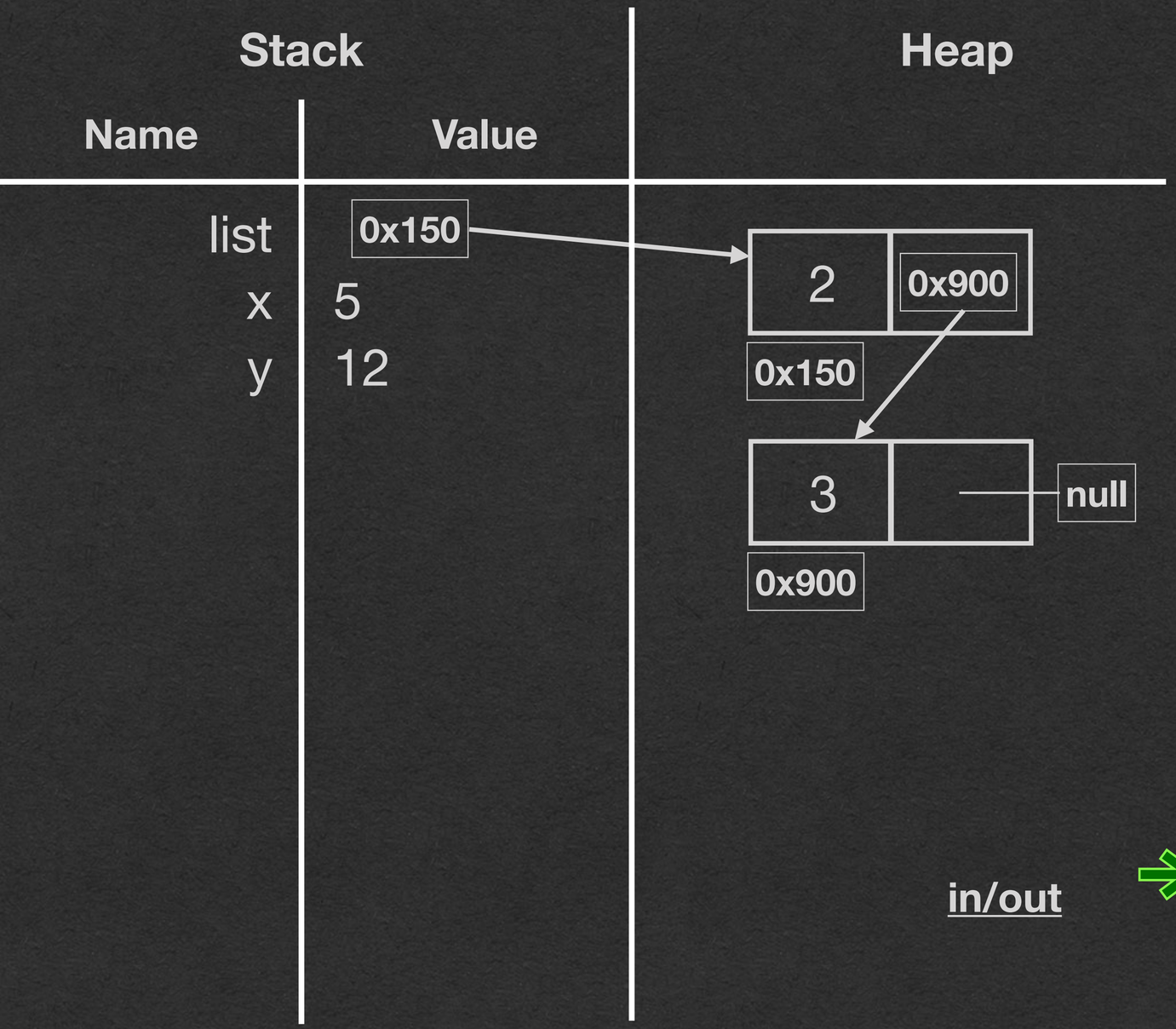
Option 2

- 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**

in/out

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

Good Memory Heap Example



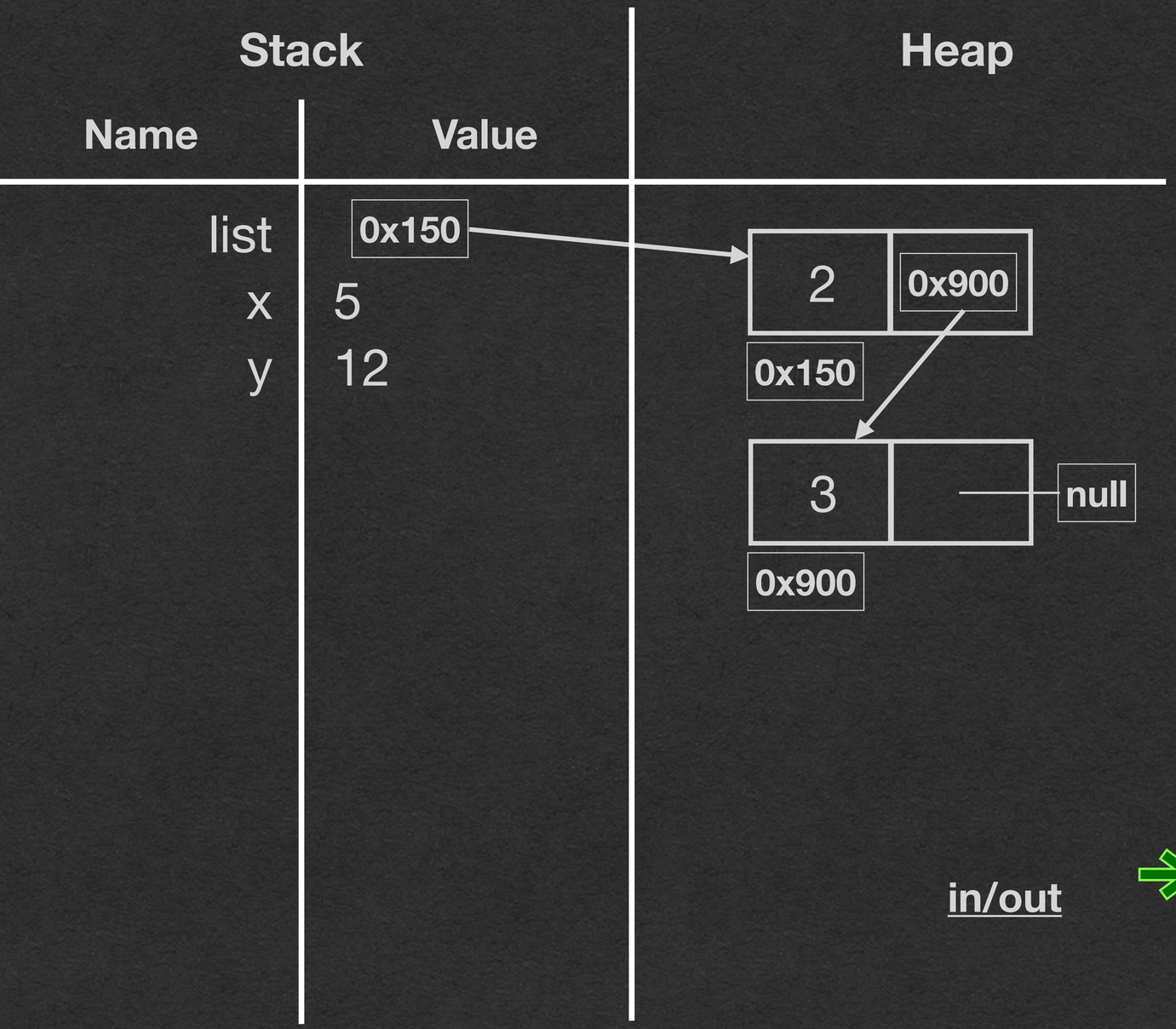
Option 3

- **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
    - This location is its **reference**

in/out

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

Good Memory Heap Example



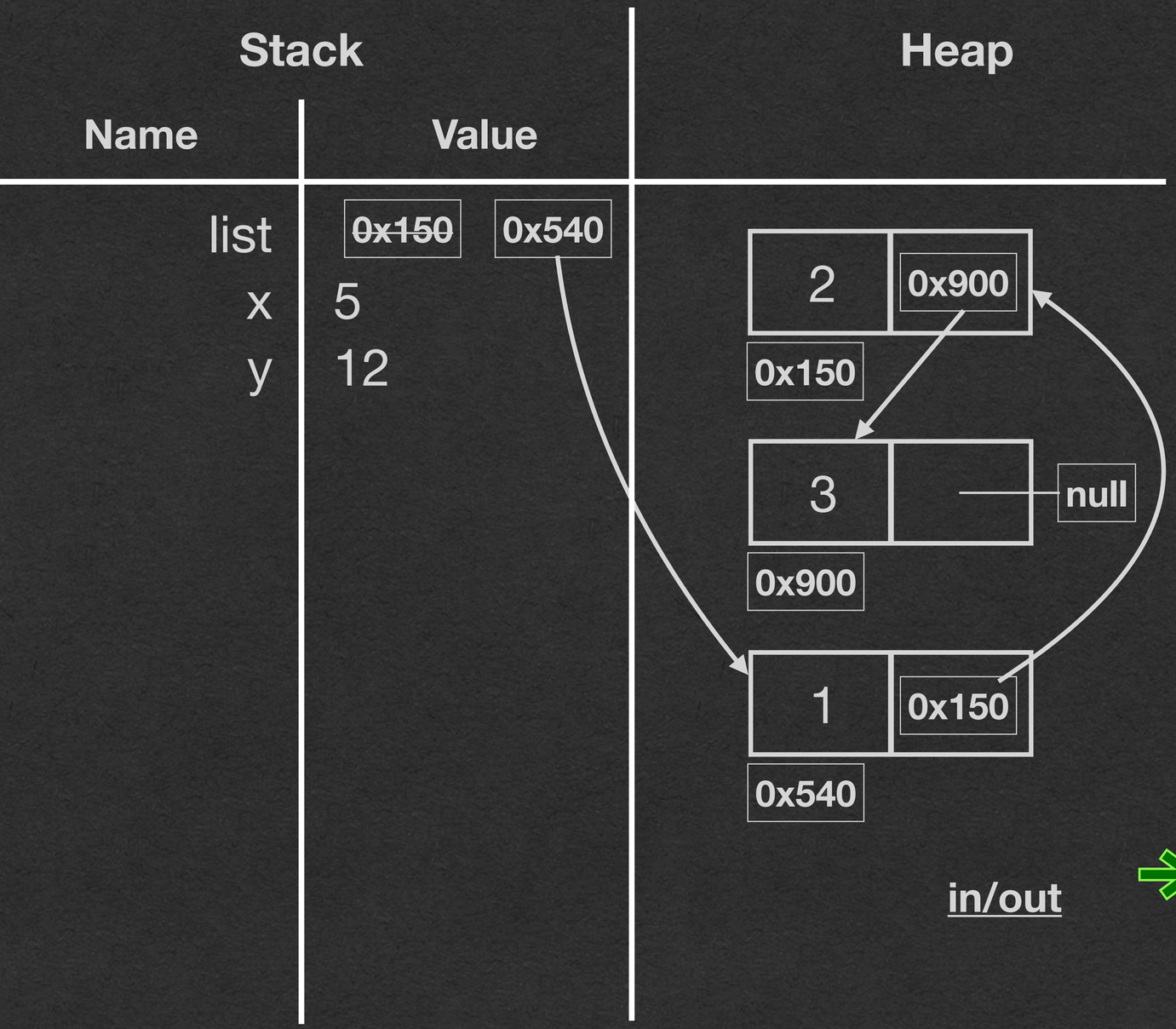
Option 3

- For lists specifically
  - Each value is stored at a different location on the heap
  - Each value has a reference to the next value
  - Last value has no reference (or a reference to null)

in/out

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

Good Memory Heap Example



Option 3

- Create a new List of Ints by adding the value 1 in heap space and have it "refer" 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 your objects**
  - Also data structures (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’s location in memory

Pass By Reference

- 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 on the heap!