

# 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\*

**\*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  
}
```

# Bad Memory Heap Example

Stack		Heap
Name	Value	
<b>list(0)</b>	<b>2</b>	
<b>list(1)</b>	<b>3</b>	

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



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	
<b>list(0)</b>	<b>2</b>	
<b>list(1)</b>	<b>3</b>	
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

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		
<b>list(0)</b>	<b>2</b>		
<b>list(1)</b>	<b>3</b>		
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	
<b>list(0)</b>	<b>1</b>	
<b>list(1)</b>	<b>2</b>	
<b>list(2)</b> x	5 <b>3</b>	
y	12	

## 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**

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	
x	5	
y	12	
<b>list(0)</b>	<b>1</b>	
<b>list(1)</b>	<b>2</b>	
<b>list(2)</b>	<b>3</b>	

## 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**

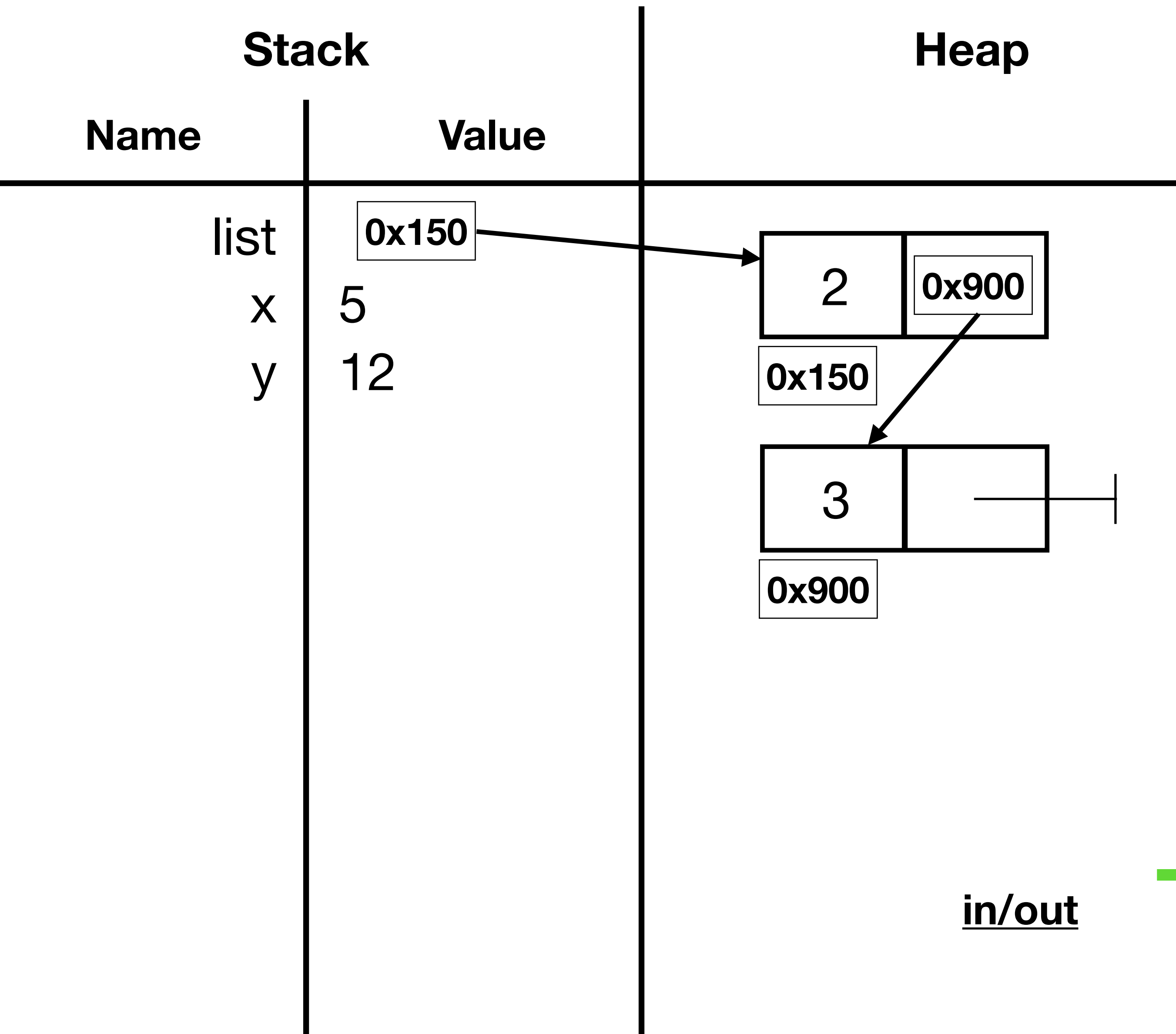
in/out



```
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

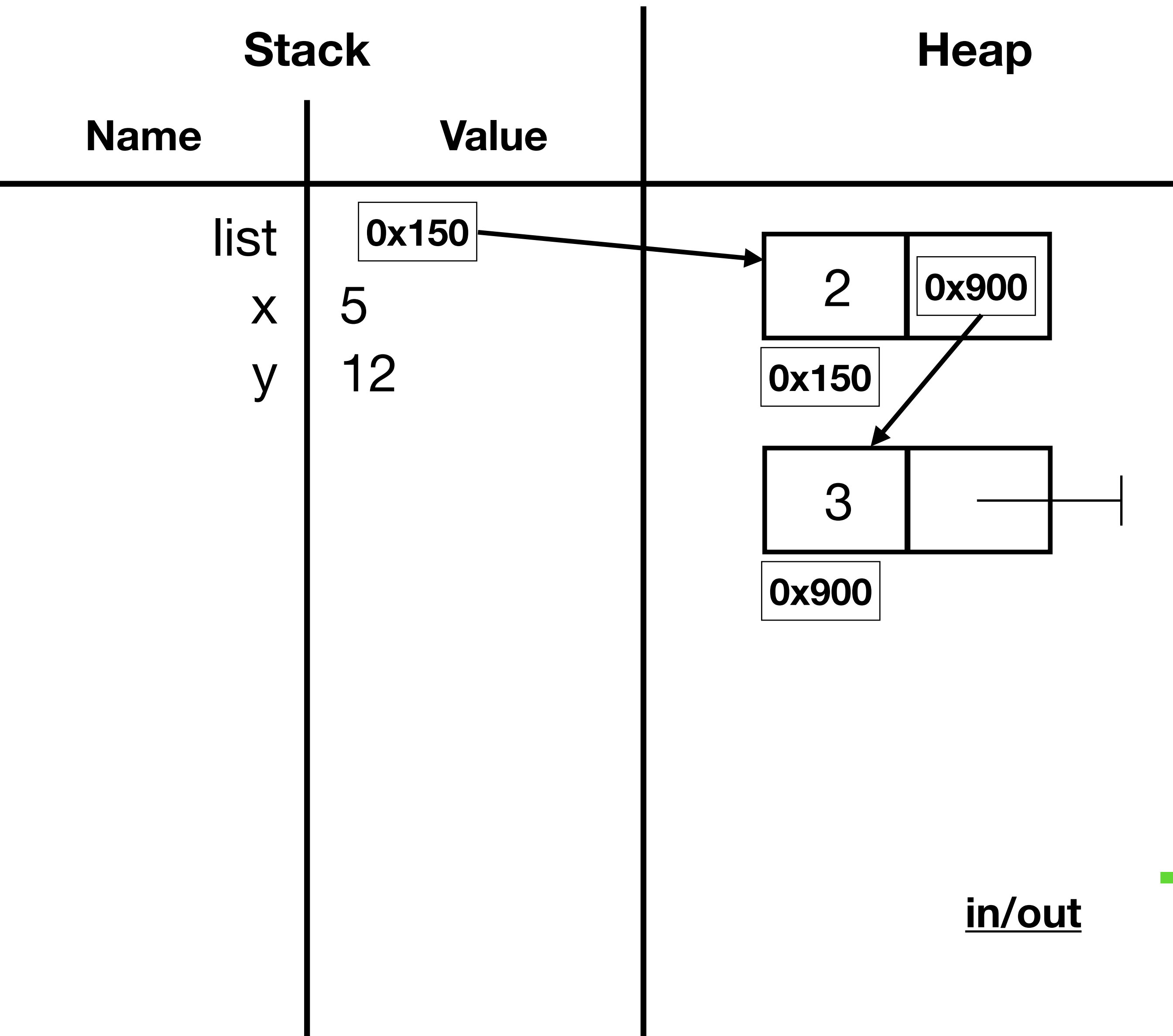
- **Allocate objects in heap memory**
- When the list is created
  - Use memory that is not part of the stack to store its values
  - Store the location, in memory, of these values in the "list" variable

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

in/out



# Good Memory Heap Example

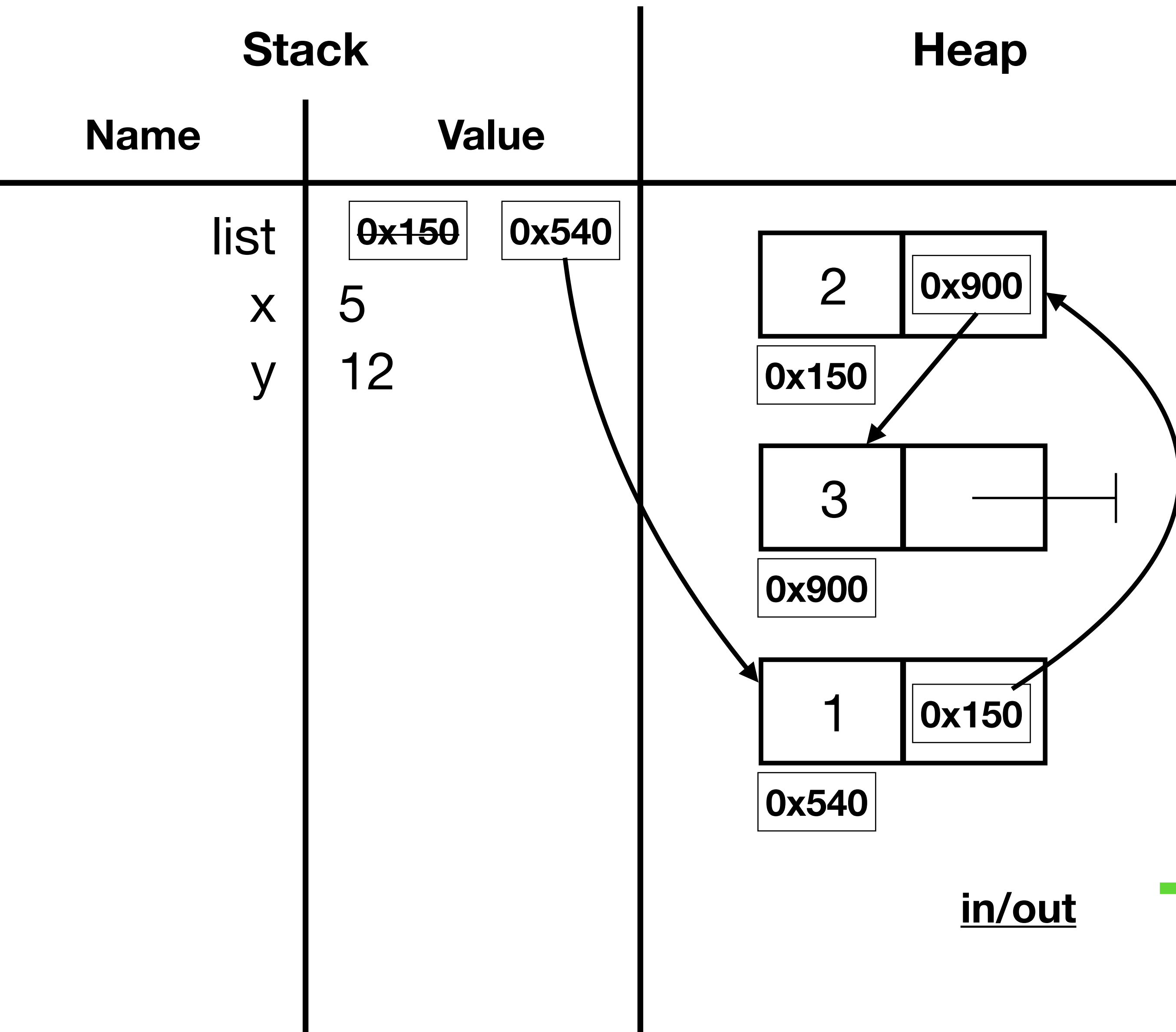


## Option 3

- For list 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)

```
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 "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 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!