Immutability

Immutable Objects

- **Values stored in state variables cannot change**

- Immutable objects are stored on the heap just like any other object

  - But we don't worry about the state changing when we pass the reference to a method/function

Immutable Objects

- What if an immutable object needs to change state?

- Create a copy of the object with the change applied

Immutable Objects

- This ImmutableCounter class takes an initial value in its constructor and has methods to increment and decrement this value

- The internal Int is a value and cannot change

  - It also can't be accessed (Artificial restriction to show more recursion)

```scala
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
 }

 def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
 }

 def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
 }

}
```

Immutable Objects

- Since the Int cannot change

  - We simulate changes by creating a new object on the heap with the change applied

- Create and return a new ImmutableCounter whenever a "change" is made

```scala
class ImmutableCounter(counter: Int) {

  def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

Immutable Objects

- Since we return a new ImmutableCounter

  - We must use this return value or we will not see the change

```scala
class ImmutableCounter(counter: Int) {

  def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```scala
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Immutable Objects

- What if we want to increment this object 10 times?

- Since we [artificially] restrict access to the Int we can only increment and decrement

- We could use a loop and reassign a variable at each iteration (requires var)

```scala
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

```scala
class ImmutableCounter(counter: Int) {

 def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

Immutable Objects

- What if we want to increment this object 10 times?

- Use a recursive approach

  - Base case of n==0

  - Recursively increment/decrement and make a recursive call with n closer to 0

```
class ImmutableCounter(counter: Int) {

  def printCount():Unit = {
    println(this.counter)
  }

  def increase(): ImmutableCounter = {
    new ImmutableCounter(this.counter + 1)
  }

  def decrease(): ImmutableCounter = {
    new ImmutableCounter(this.counter - 1)
  }

}
```

```
def updateCounter(n: Int, counter: ImmutableCounter): ImmutableCounter = {
  if(n==0){
    counter
  }else if(n < 0){
    updateCounter(n+1, counter.decrease())
  }else{
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: ImmutableCounter = new ImmutableCounter(10)
  val counter2: ImmutableCounter = updateCounter(20, counter)

  counter.printCount()
  counter2.printCount()
}
```

Scala Lists are Immutable

Scala Lists

- Scala Lists cannot be modified in any way

  - They are immutable

- If an element is added/removed *or* any value is changed

  - A new list is created

- Must store the new list in a variable to see the change

- With "var" you need to reassign the new List to the variable to see the change

```scala
def main(args: Array[String]): Unit = {
    val list: List[Int] = List(2)
    val newList: List[Int] = 1 :: list
}
```

```scala
def main(args: Array[String]): Unit = {
    var list: List[Int] = List(2)
    list: List[Int] = 1 :: list
}
```

Memory Diagram

```scala
class Counter(counter: Int) {
 def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
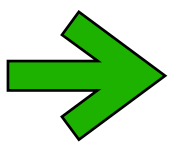
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```
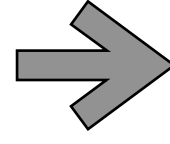
- Slightly modified code for space concerns

Stack

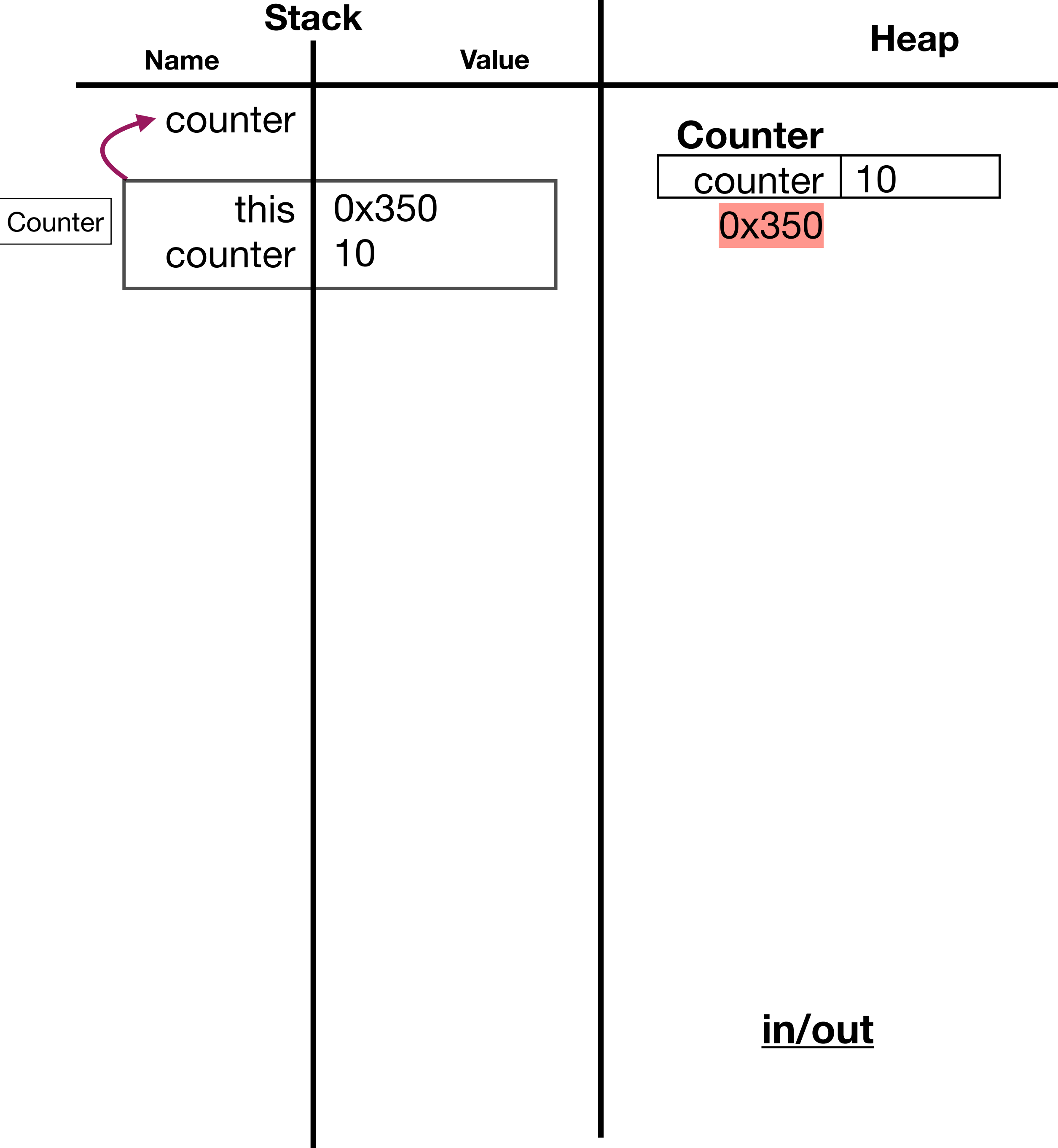| Name	Value

Heap

in/out

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```

```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

- Creating an immutable object

- Works the same as creating any other object

Stack

Name		Value	
counter			
this		0x350	
counter		10	

Counter

Heap

Counter

counter	10

0x350

in/out

```scala
class Counter(counter: Int) {
 def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
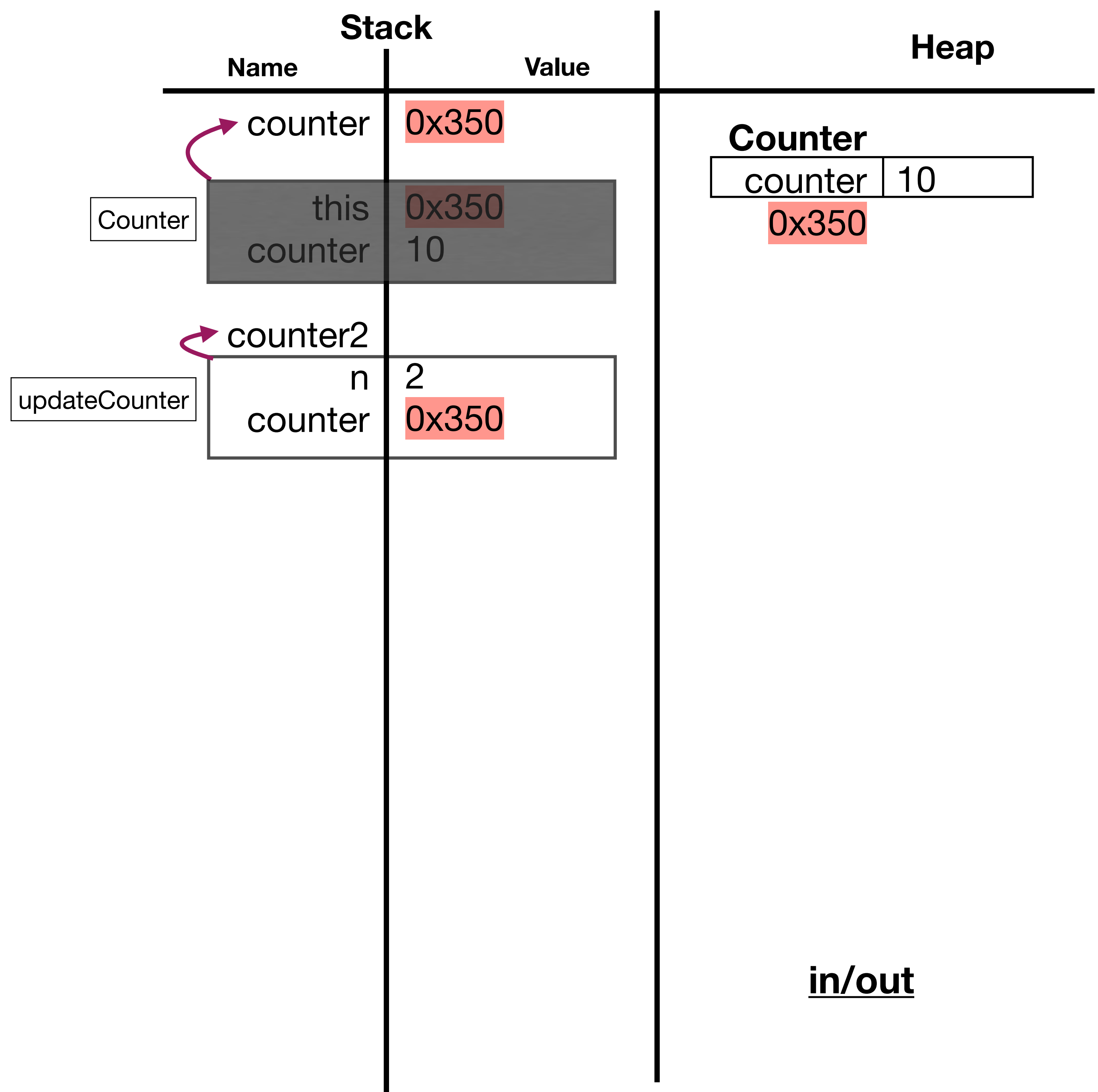
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```
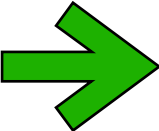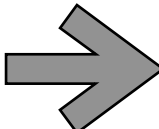
Stack

Name	Value
counter	0x350
this	0x350
counter	10
counter2	
n	2
counter	0x350

Counter

updateCounter

Heap

Counter

counter	10

0x350

in/out

- Call updateCounter

- Check for the base case

- n is >0 so we run the recursive part of the method

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
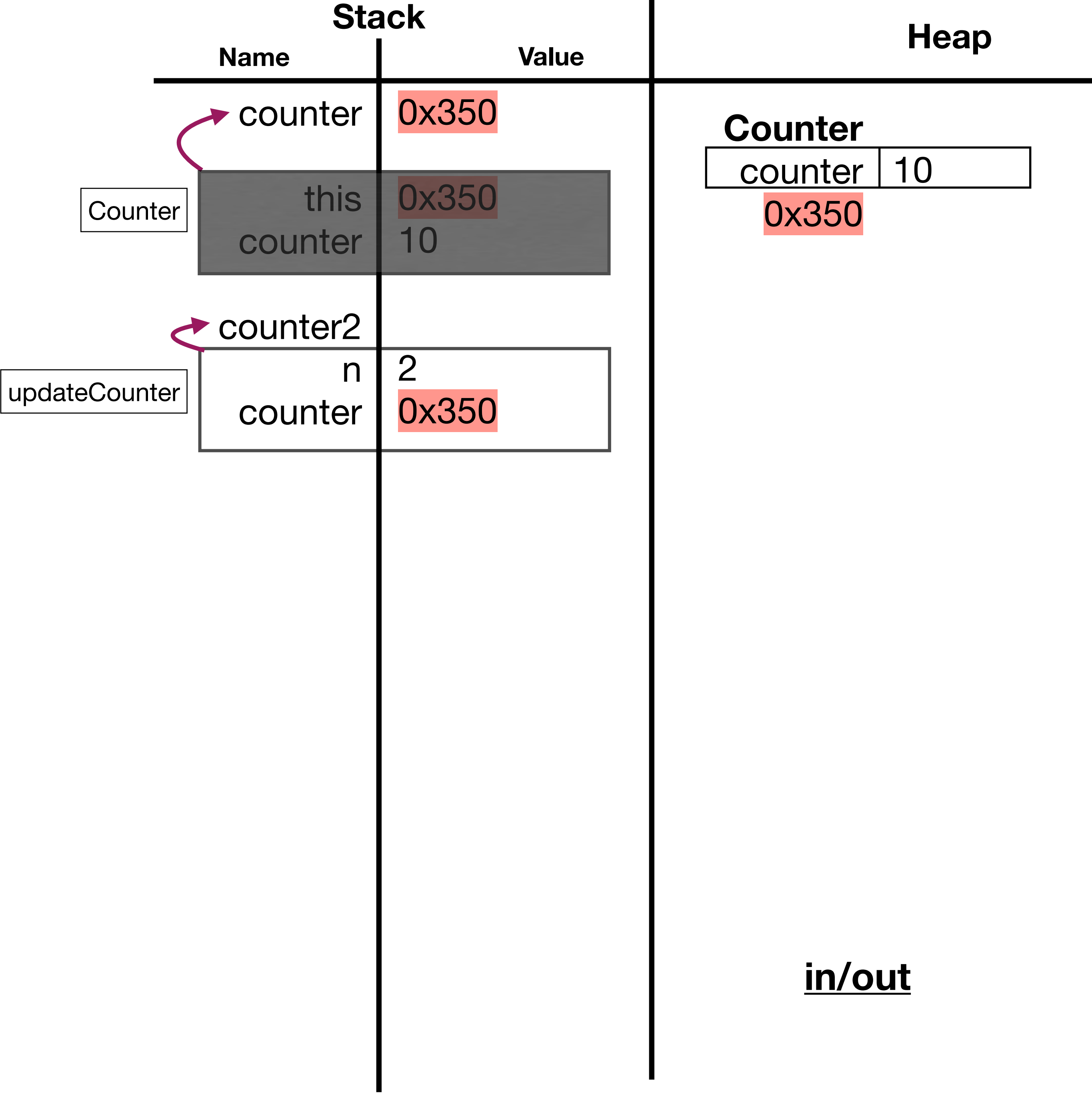
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```
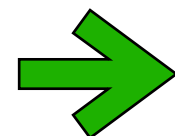
Stack

Name	Value
counter	0x350

Counter
| this | 0x350 |
| counter | 10 |

| counter2 | |

updateCounter
| n | 2 |
| counter | 0x350 |

Heap

Counter

counter	10

0x350

in/out

- Our goal is to do as little work as possible to get closer to the base case

- We eventually have to call increase twice

  - Let's call it once and make a recursive call

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```

```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```
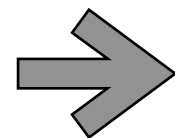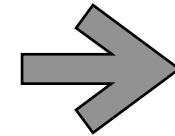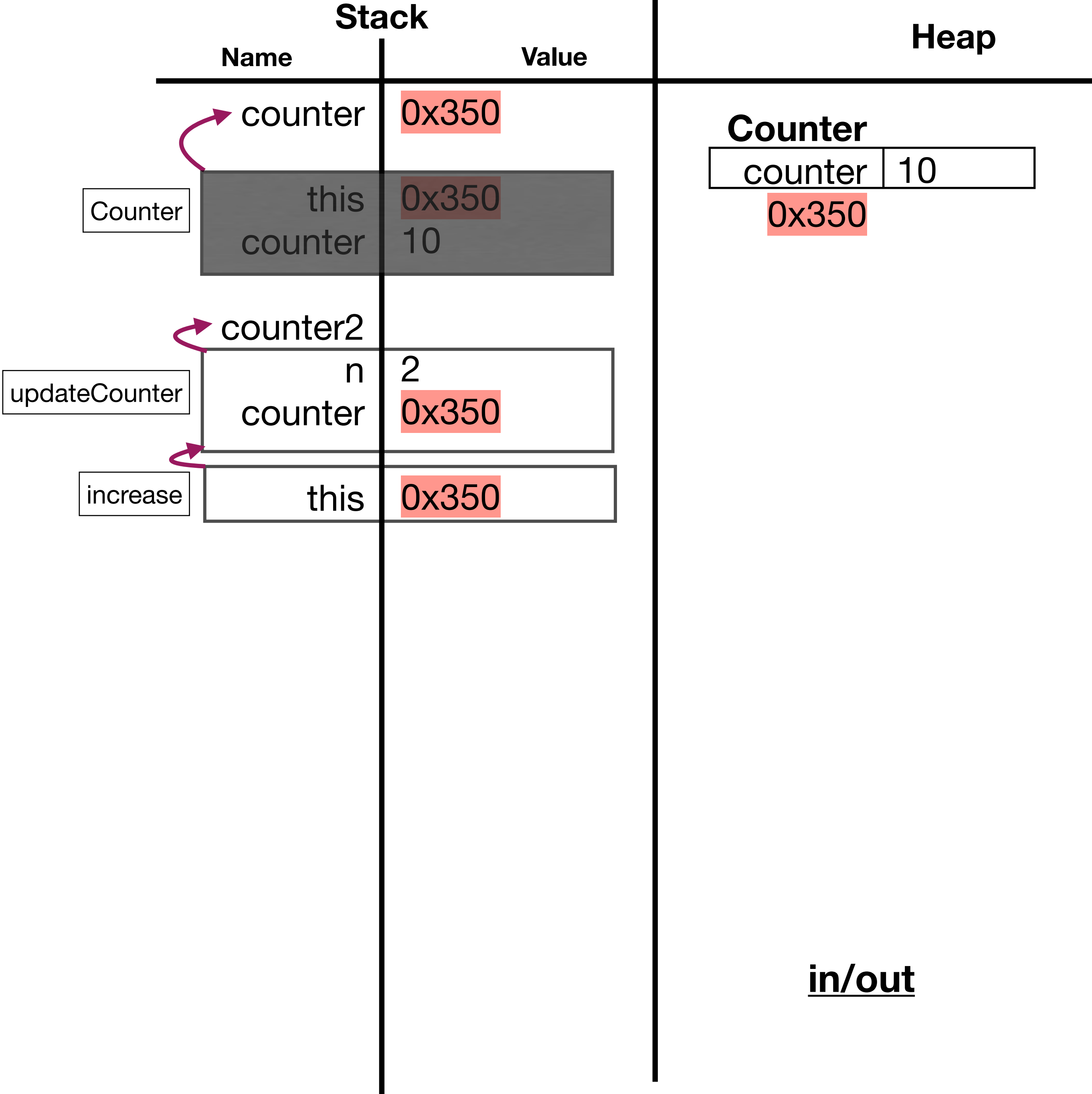
Stack

Name	Value
counter	0x350

Counter
| this | 0x350 |
| counter | 10 |

| counter2 | |

updateCounter
| n | 2 |
| counter | 0x350 |

increase
| this | 0x350 |

Heap

Counter

| counter | 10 |

0x350

in/out

- Before making the recursive call

  - call increase once

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
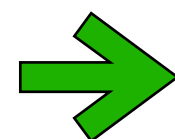
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```
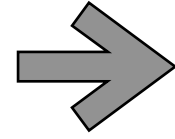
Stack

Name	Value
counter	0x350

Counter	this	0x350
	counter	10

	counter2	
updateCounter	n	2
	counter	0x350

increase	this	0x350

Counter	this	0x200
	counter	11

updateCounter	n	1
	counter	0x200

Heap

Counter

counter	10

0x350

Counter

counter	11

0x200

in/out

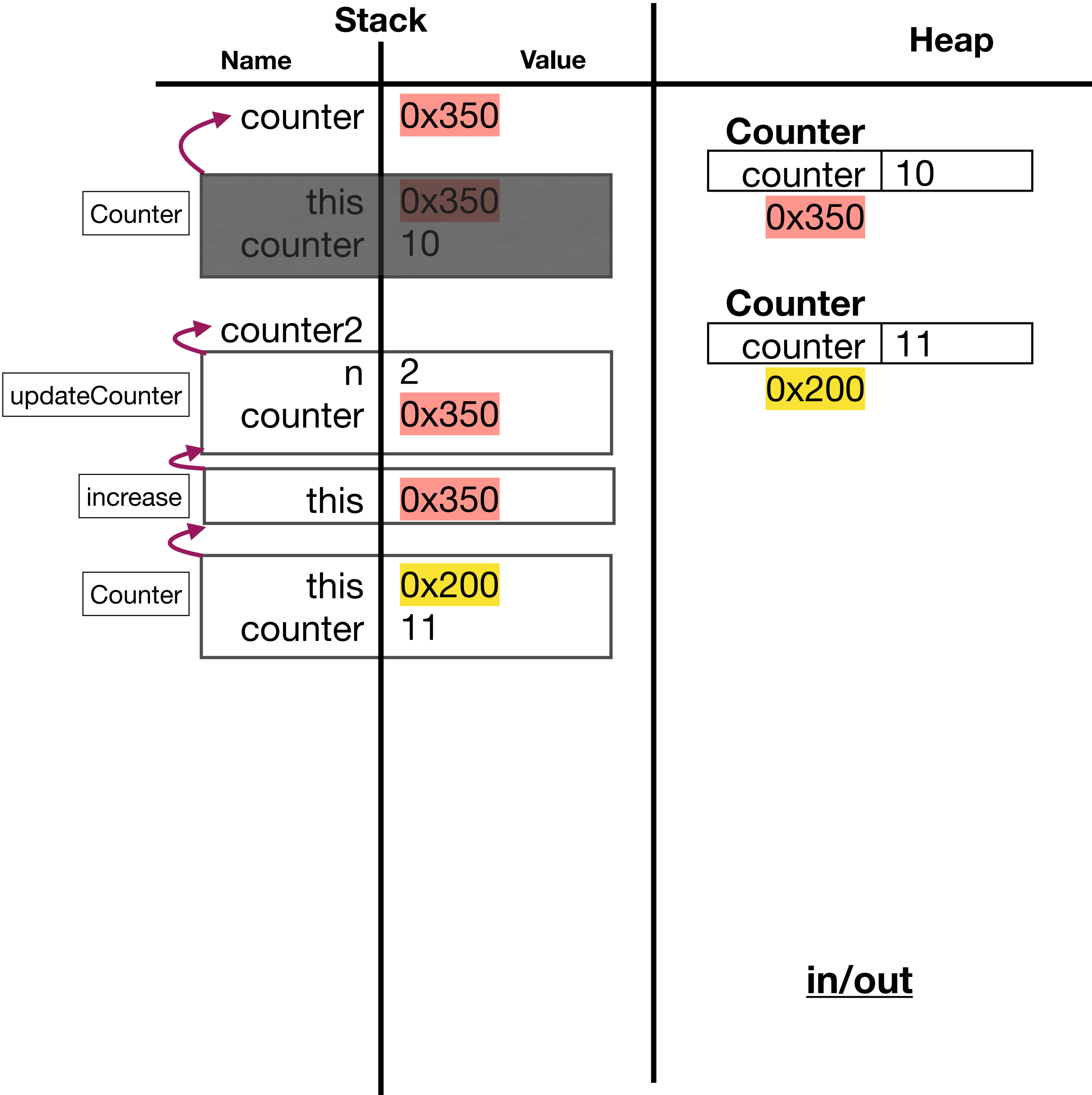- Methods return and we're ready for the recursive call

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
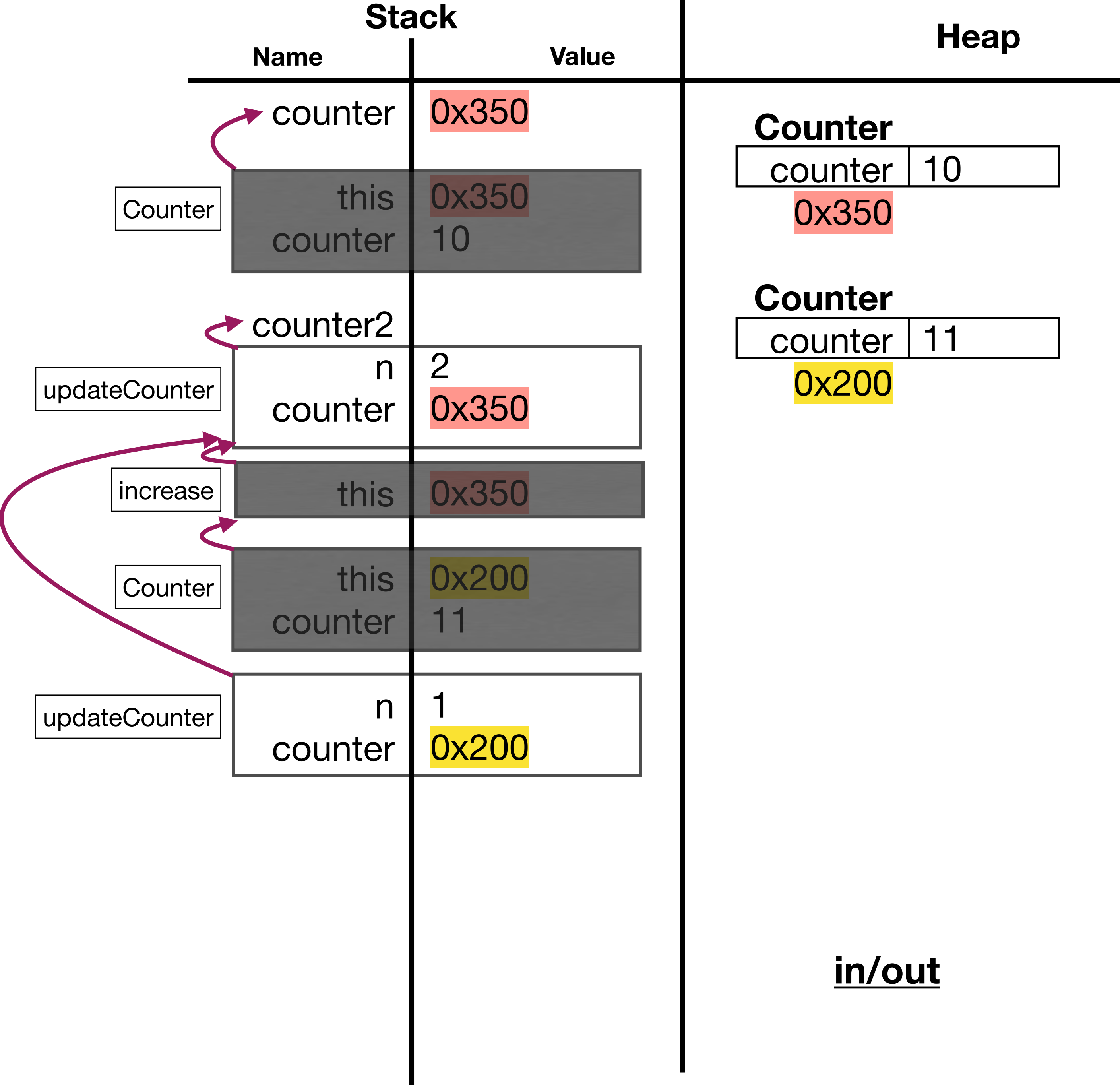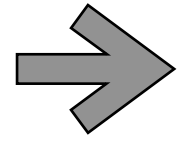
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

- Repeat the process since we have still not reached the base case

Stack

Name	Value
counter	0x350

Counter
| this | 0x350 |
| counter | 10 |

| counter2 | |

updateCounter
| n | 2 |
| counter | 0x350 |

increase
| this | 0x350 |

Counter
| this | 0x200 |
| counter | 11 |

updateCounter
| n | 1 |
| counter | 0x200 |

increase
| this | 0x200 |

Counter
| this | 0x480 |
| counter | 12 |

Heap

Counter
| counter | 10 |

0x350

Counter
| counter | 11 |

0x200

Counter
| counter | 12 |

0x480

in/out

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
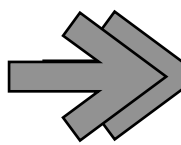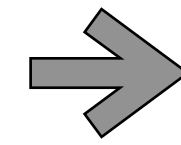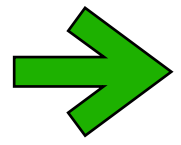
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

- We've reached the base case

- If n is 0, return the provided counter

Stack

Name		Value
	counter	0x350
Counter	this	0x350
	counter	10
	counter2	
updateCounter	n	2
	counter	0x350
increase	this	0x350
Counter	this	0x200
	counter	11
updateCounter	n	1
	counter	0x200
increase	this	0x200
Counter	this	0x480
	counter	12
updateCounter	n	0
	counter	0x480

Heap

Counter

counter	10

0x350

Counter

counter	11

0x200

Counter

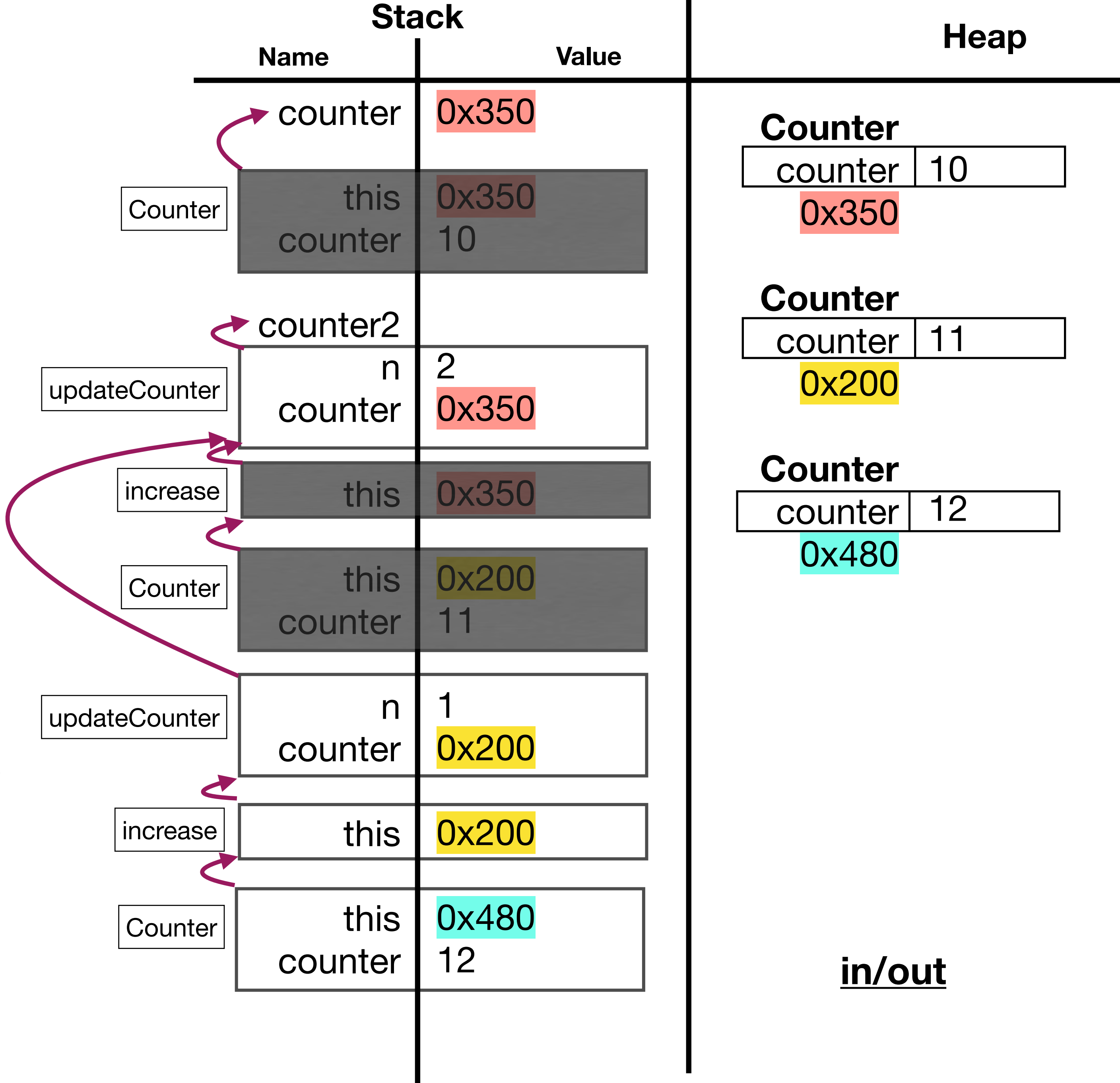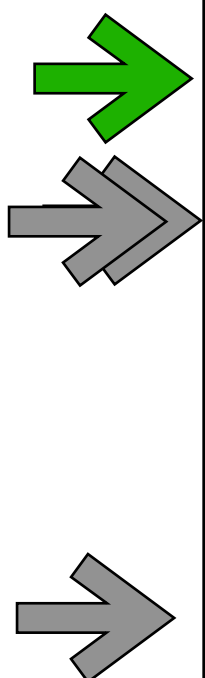| counter	12

0x480

in/out

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
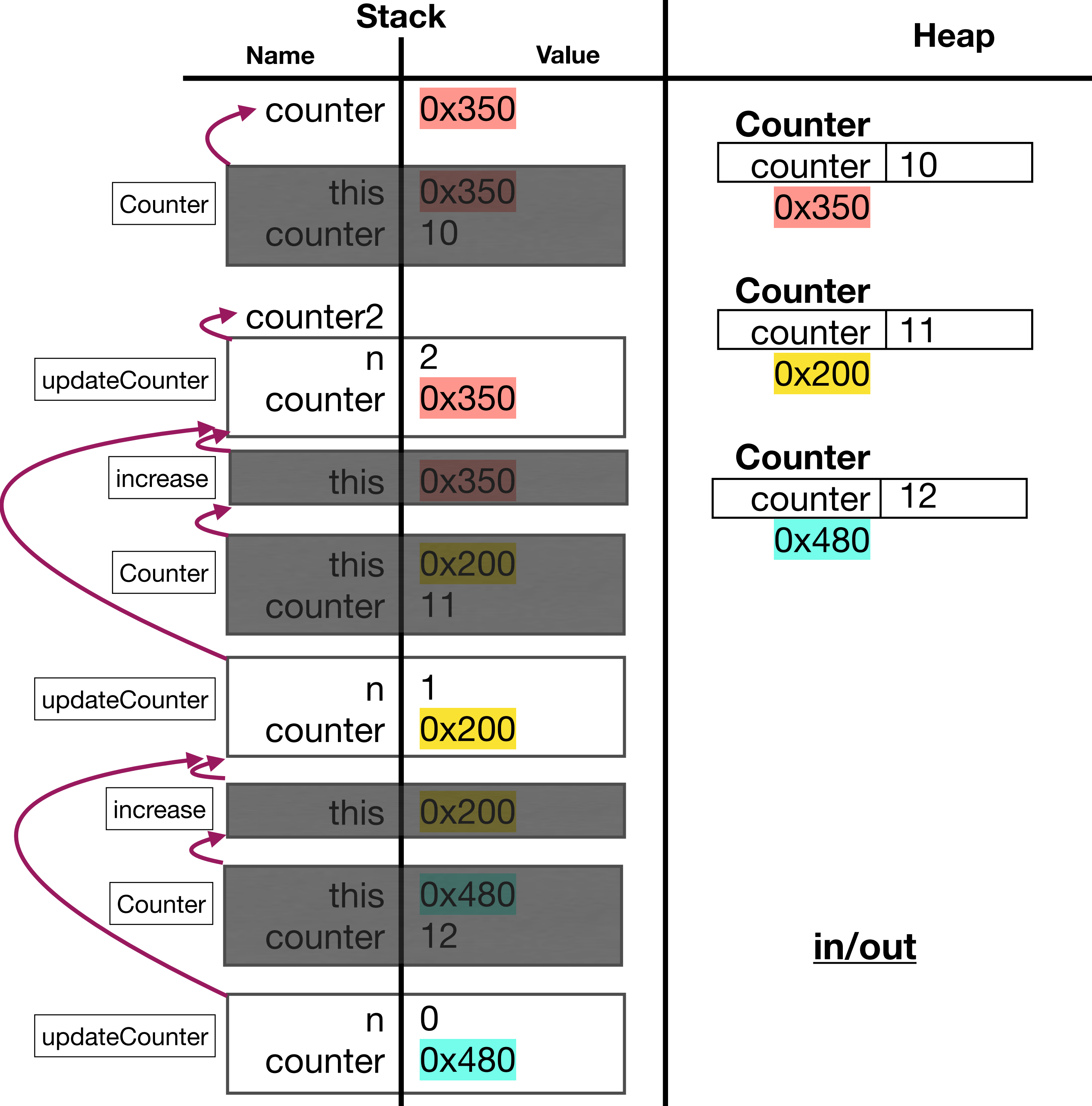
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

• Return the reference 0x480 all the way up the recursive calls

Stack

Name	Value
counter	0x350

Counter

this	0x350
counter	10

counter2	0x480

updateCounter

n	2
counter	0x350

increase

this	0x350

Counter

this	0x200
counter	11

updateCounter

n	1
counter	0x200

increase

this	0x200

Counter

this	0x480
counter	12

updateCounter

n	0
counter	0x480

Heap

Counter

counter	10

0x350

Counter

counter	11

0x200

Counter

counter	12

0x480

in/out

```scala
class Counter(counter: Int) {
  def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
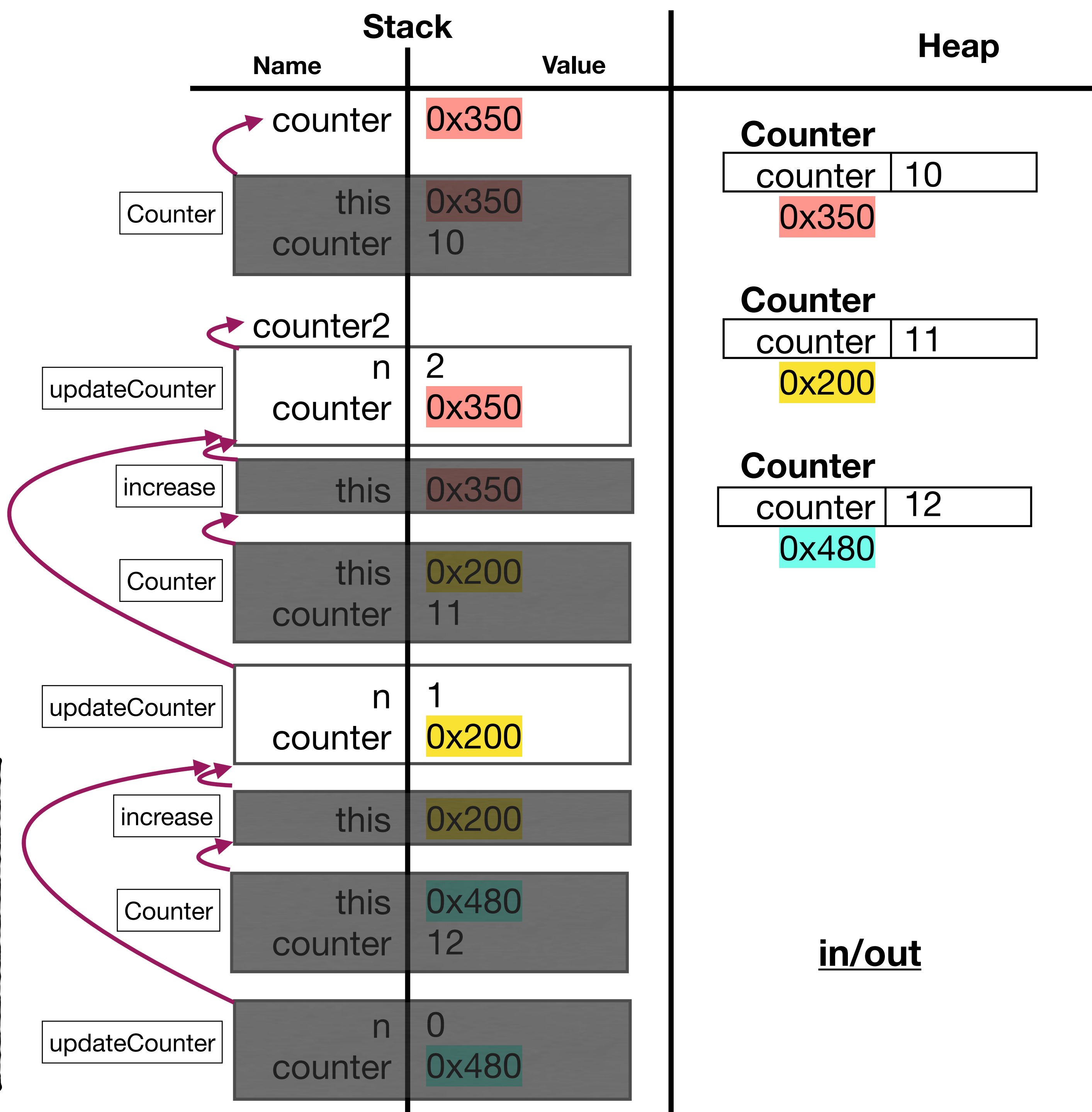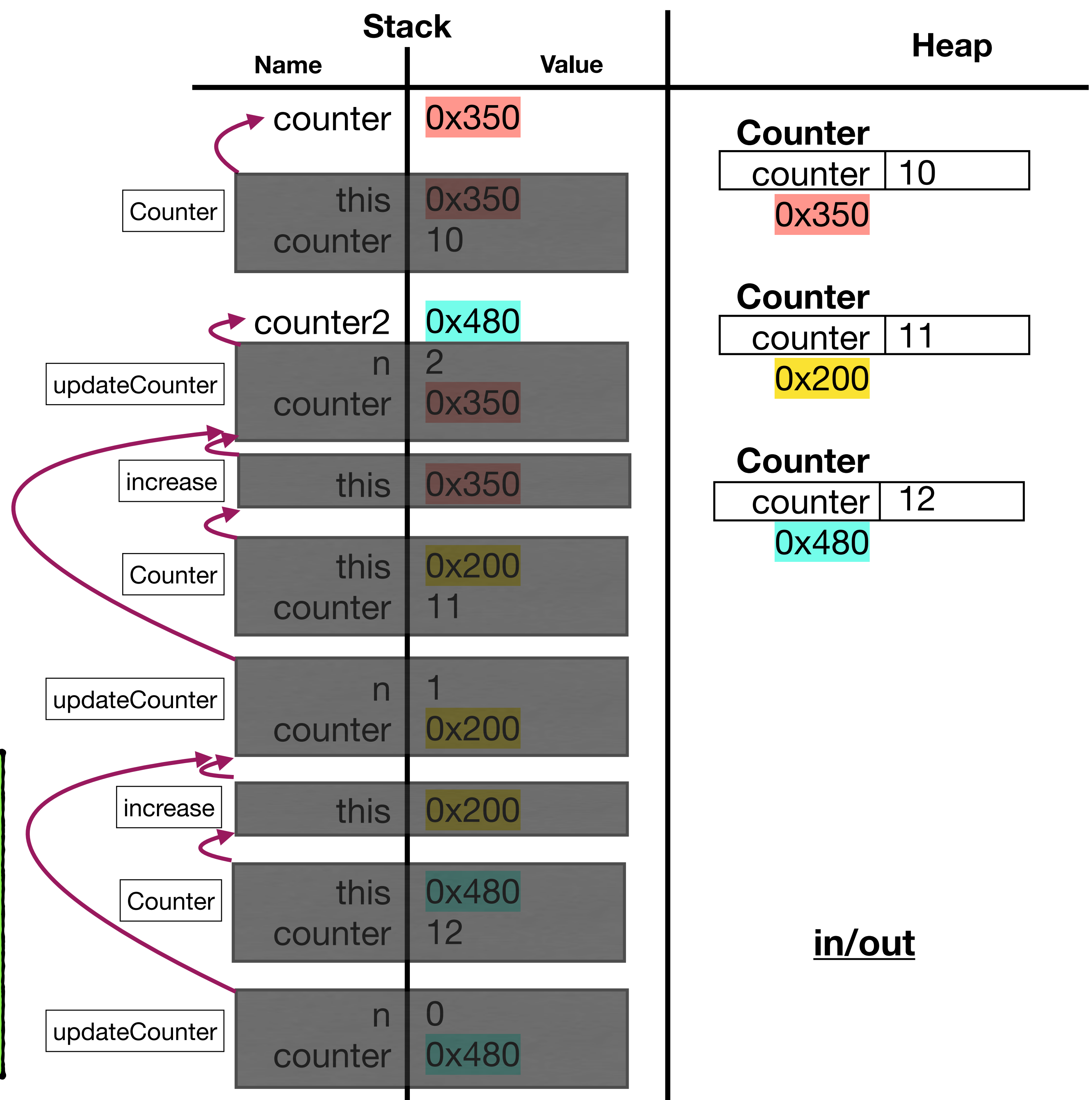
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

Stack

Name	Value
counter	0x350

Counter	this	0x350
	counter	10

	counter2	0x480
updateCounter	n	2
	counter	0x350

increase	this	0x350

Counter	this	0x200
	counter	11

updateCounter	n	1
	counter	0x200

increase	this	0x200

Counter	this	0x480
	counter	12

updateCounter	n	0
	counter	0x480

Heap

Counter

counter	10

0x350

Counter

counter	11

0x200

Counter

counter	12

0x480

in/out

10

12

- Print the values to the screen
- counter has a value 10
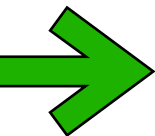- counter2 has a value 12

```scala
class Counter(counter: Int) {
 def printCount():Unit = {
    println(this.counter)
  }
  def increase(): Counter = {
    new Counter(this.counter + 1)
  }
}
```
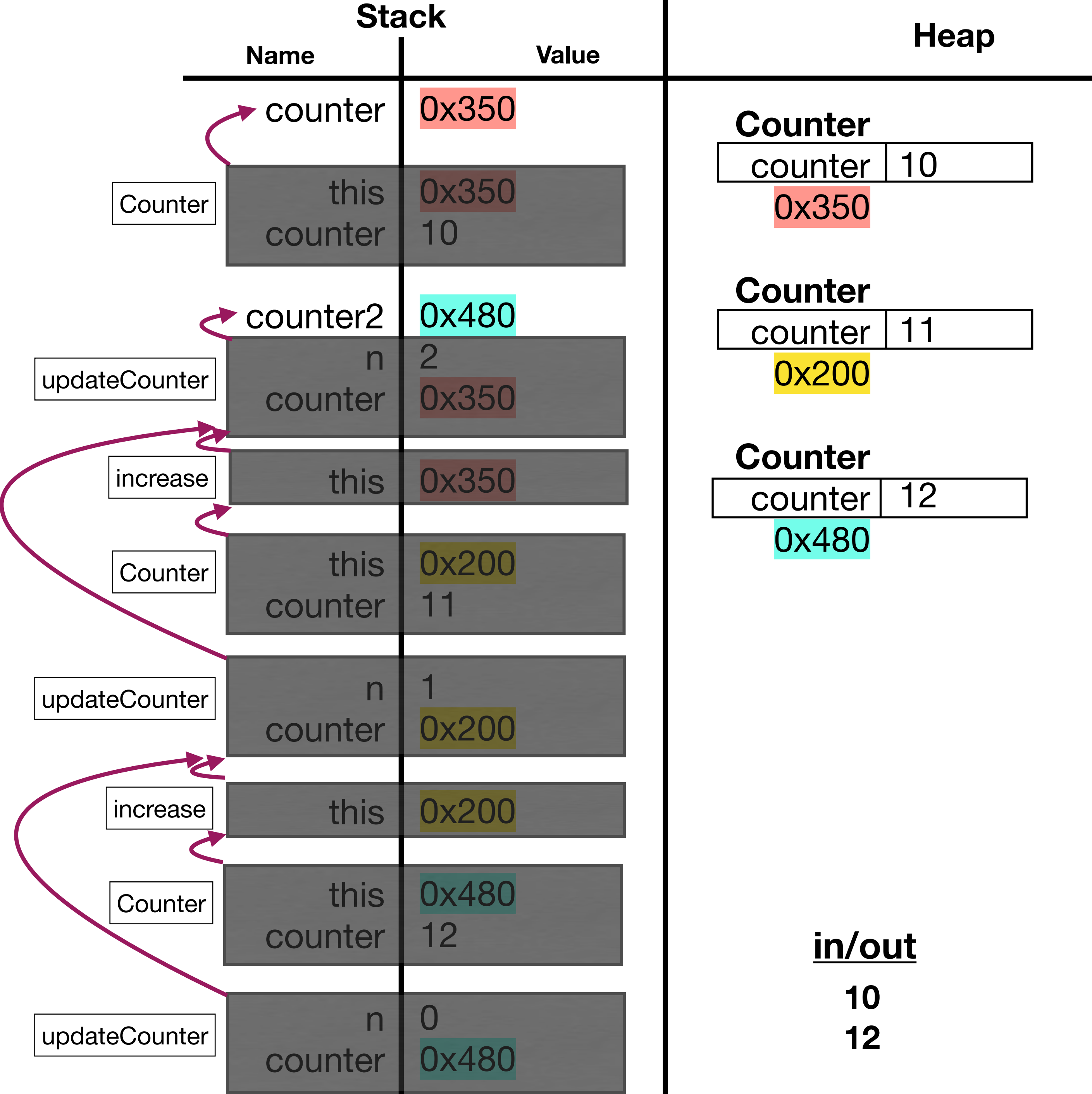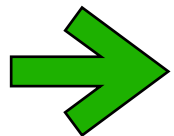
```scala
def updateCounter(n: Int, counter: Counter): Counter = {
  if(n<=0) {
    counter
  } else {
    updateCounter(n-1, counter.increase())
  }
}

def main(args: Array[String]): Unit = {
  val counter: Counter = new Counter(10)
  val counter2: Counter = updateCounter(2, counter)

  counter.printCount()
  counter2.printCount()
}
```

- Note: Each time we needed to change a Counter, we created a new Counter object

Stack

Name		Value
	counter	0x350
Counter	this	0x350
	counter	10
	counter2	0x480
updateCounter	n	2
	counter	0x350
increase	this	0x350
Counter	this	0x200
	counter	11
updateCounter	n	1
	counter	0x200
increase	this	0x200
Counter	this	0x480
	counter	12
updateCounter	n	0
	counter	0x480

Heap

Counter

counter	10

0x350

Counter

counter	11

0x200

Counter

counter	12

0x480

in/out

10

12