Programming Languages and Techniques (CIS1200)

Lecture 24

Resizable Arrays, Java ASM

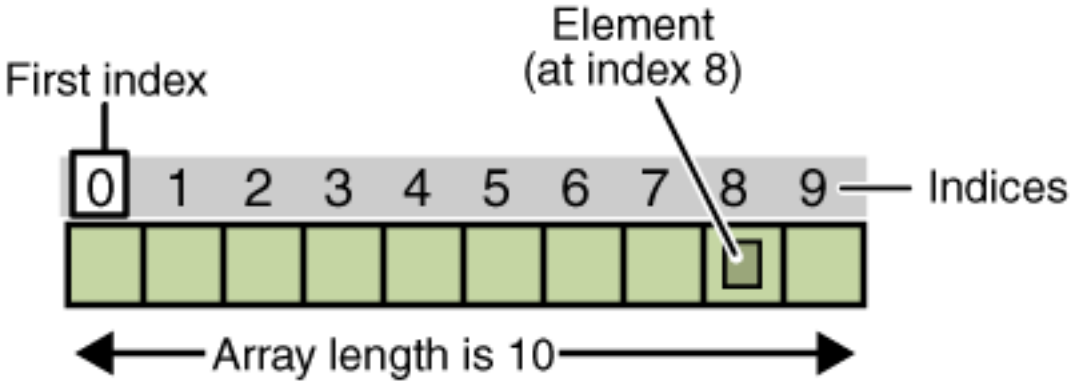
Chapters 22 and 23

Announcements

- HW06: Pennstagram
  - Java array programming
  - Due *Thursday*, October 31st at 11.59pm

Recap: Java Arrays

- An array is a sequentially ordered collection of values that can be indexed in *constant* time
- Index elements from 0



- Basic array expression forms

`a[i]`	access element of array `a` at index `i`
`a[i] = e`	assign `e` to element of array `a` at index `i`
`a.length`	get the number of elements in `a`

Multidimensional Arrays

Multi-Dimensional Arrays

A 2-d array is just an array of arrays...

```
String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
        {"Smith", "Jones"}};

System.out.println(names[0][0] + names[1][0]);
//  --> Mr. Smith
System.out.println(names[0][2] + names[1][1]);
//  --> Ms. Jones
```

`String[][]` just means `(String[])[]`

`names[1][1]` just means `(names[1])[1]`

More brackets → more dimensions

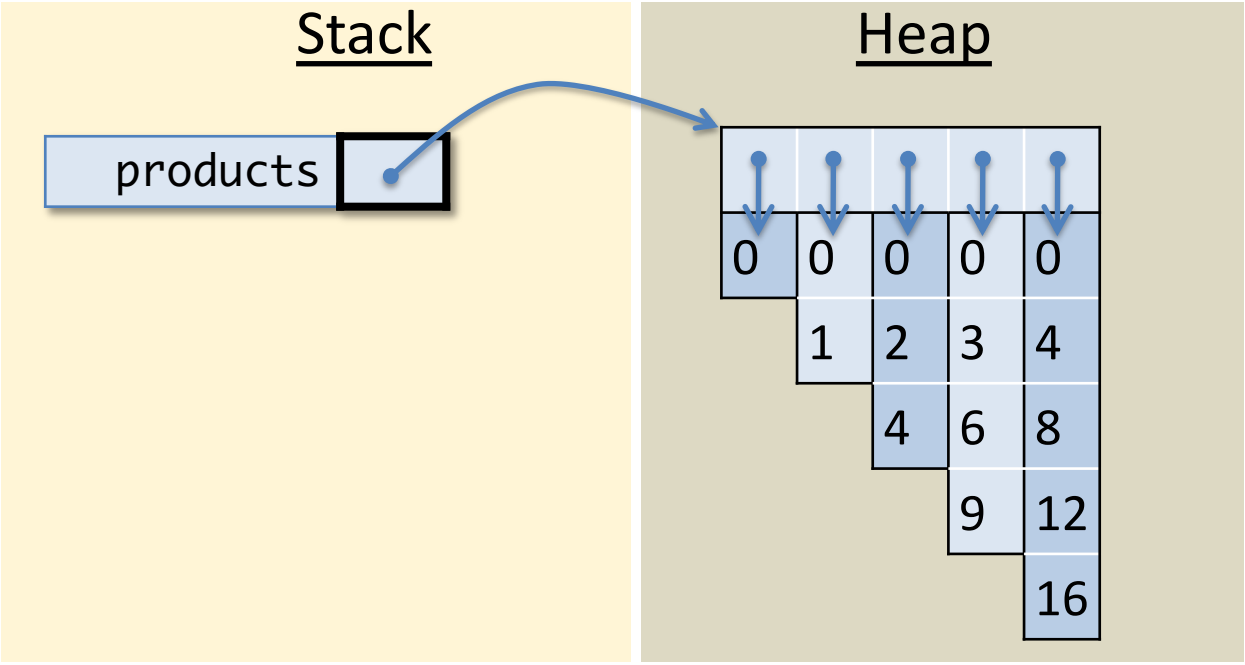
Multi-Dimensional Arrays

```
int[][] products = new int[5][];
for (int col = 0; col < 5; col++) {
    products[col] = new int[col + 1];
    for (int row = 0; row <= col; row++) {
        products[col][row] = col * row;
    }
}
```

What would a “Java ASM” stack and heap look like after running this program?

Multi-Dimensional Arrays

```
int[][] products = new int[5][];
for (int col = 0; col < 5; col++) {
    products[col] = new int[col + 1];
    for (int row = 0; row <= col; row++) {
        products[col][row] = col * row;
    }
}
```



Note: This heap picture is simplified – it omits the class identifiers and length fields for all 6 of the arrays depicted. (Contrast with the array shown earlier.)

Note also that orientation doesn't matter on the heap.

Design Exercise: Resizable Arrays

Arrays that grow without bound.

Please see Chapter 32 in the Lecture Notes for more practice with arrays

Object encapsulation

- ***All modification to the state of the object must be done using the object's own methods.***

- Use encapsulation to preserve invariants about the state of the object.

- Enforce encapsulation by not returning aliases from methods.

Revenge of the Son of the Abstract Stack Machine

Java Abstract Stack Machine

- Similar to OCaml Abstract Stack Machine
  - Workspace
    - Contains the currently executing code
  - Stack
    - Remembers the values of local variables and "what to do next" after function/method calls
  - Heap
    - Stores reference types: objects and arrays

- Key differences:
  - Everything, including stack slots, is mutable by default
  - Objects store *what class was used to create them*
  - *Arrays store type information and length*
  - *New component: Class table (coming soon)*

Java Primitive Values

- The values of these data types occupy (less than) one machine word and are stored directly in the stack slots.

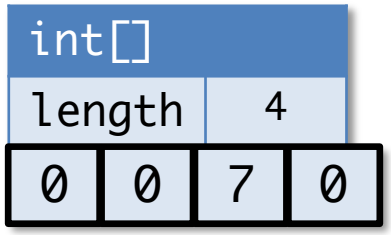
Type	Description	Values
`byte`	8-bit	`-128 to 127`
`short`	16-bit integer	`-32768 to 32767`
`int`	32-bit integer	-2^{31} to $2^{31} - 1$
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit IEEE floating point	
`double`	64-bit IEEE floating point	
`boolean`	true or false	`true false`
`char`	16-bit unicode character	`'a' 'b' '\u0000'`

Heap Reference Values

Arrays

- Type of values that it stores
- Length
- Values for all of the array elements

```
int [] a =
    { 0, 0, 7, 0 };
```

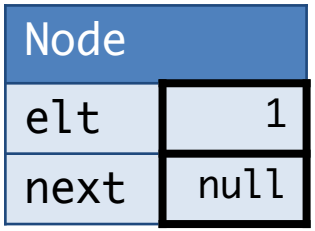


length *never* mutable; elements *always* mutable

Objects

- Name of the class that constructed it
- Values for all non-static fields

```
class Node {
   private int elt;
   private Node next;
   …
}
```



fields may or may not be mutable public/private not tracked by ASM

Objects on the ASM

What does the heap look like at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

What does the ASM look like at the end of this program?

```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

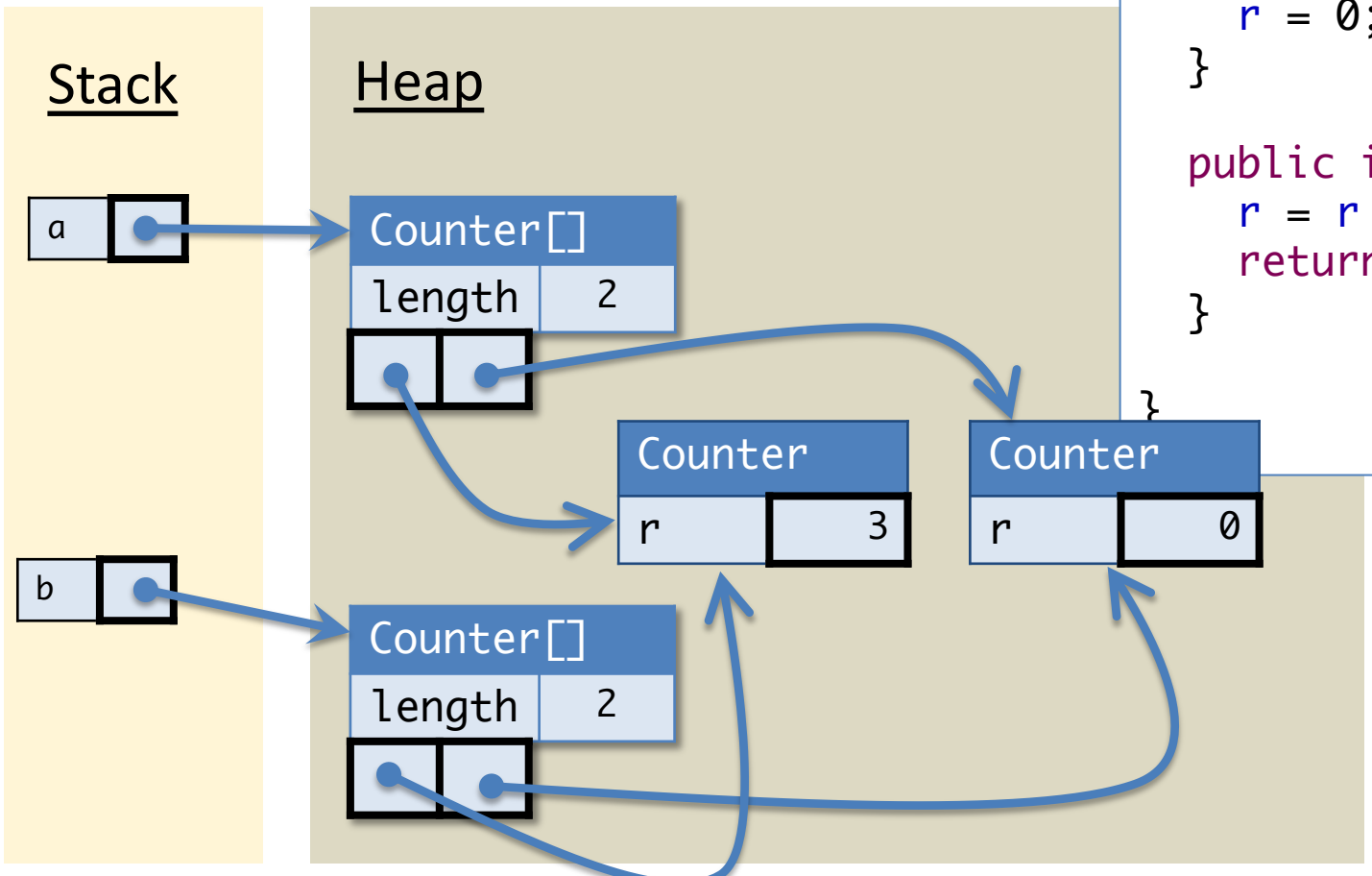
```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```



24: What does the following program print?

0

```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}
public class Test {
  public static void main(String[] args) {
    Node n1 = new Node(1,null);
    Node n2 = new Node(2,n1);
    Node n3 = n2;
    n3.next.next = n2;
    Node n4 = new Node(4,n1.next);
    n2.next.elt = 9;
    System.out.println(n1.elt);
  }

}
```

1 0%

2 0%

3 0%

4 0%

5 0%

6 0%

7 0%

8 0%

9 0%

NullPointerException 0%

What does the following program print?

1 – 9

or 10 for "NullPointerException"

```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}
public class Test {
  public static void main(String[] args) {
    Node n1 = new Node(1,null);
    Node n2 = new Node(2,n1);
    Node n3 = n2;
    n3.next.next = n2;
    Node n4 = new Node(4,n1.next);
    n2.next.elt = 9;
    System.out.println(n1.elt);
  }

}
```

Answer: 9

Workspace

```
Node n1 = new Node(1,null);
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Stack

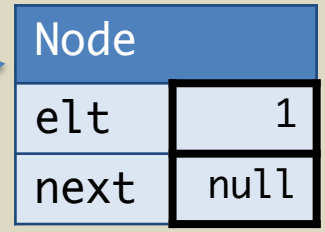
Heap

Workspace

```
Node n1 = ;
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Stack

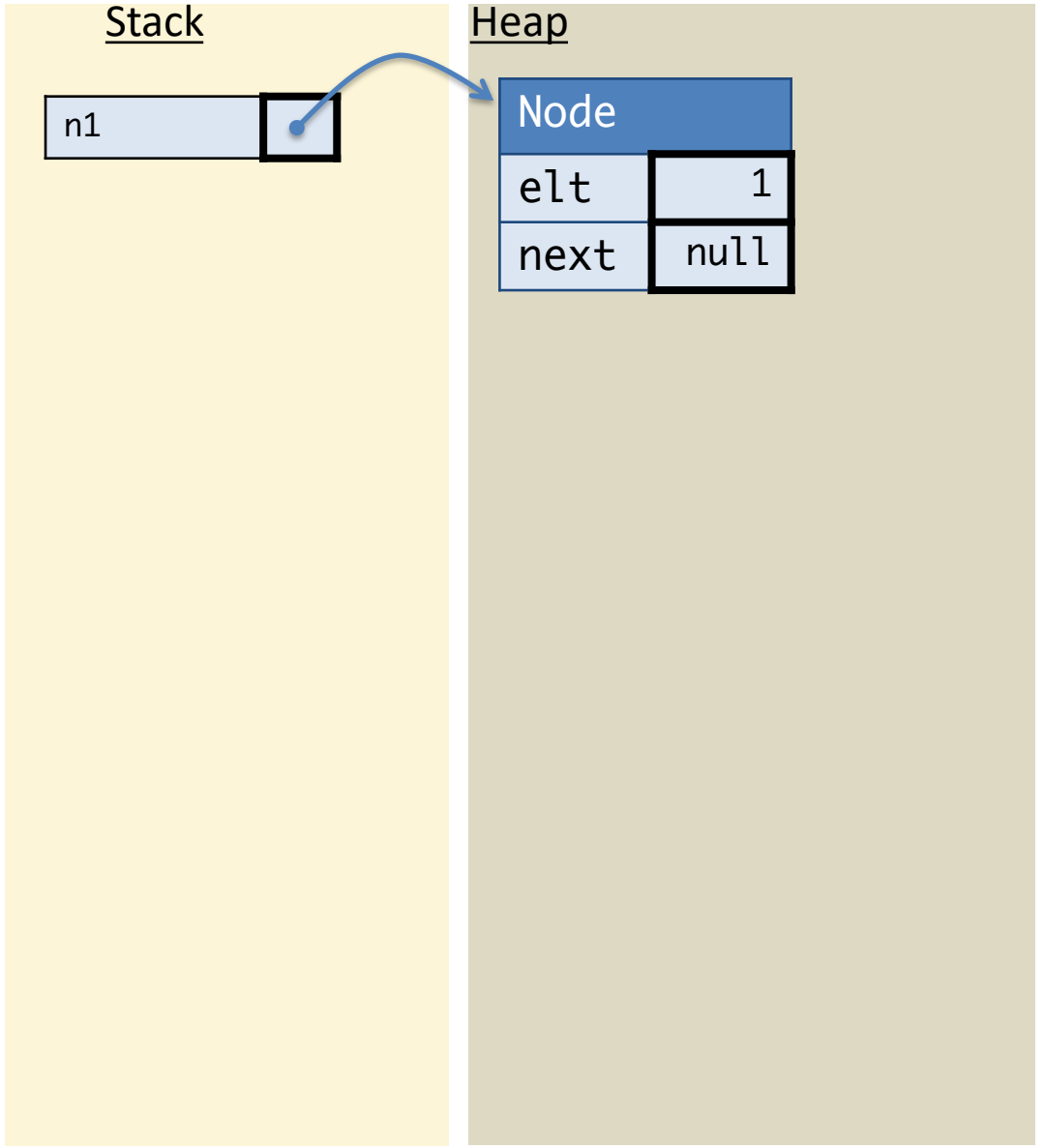
Heap

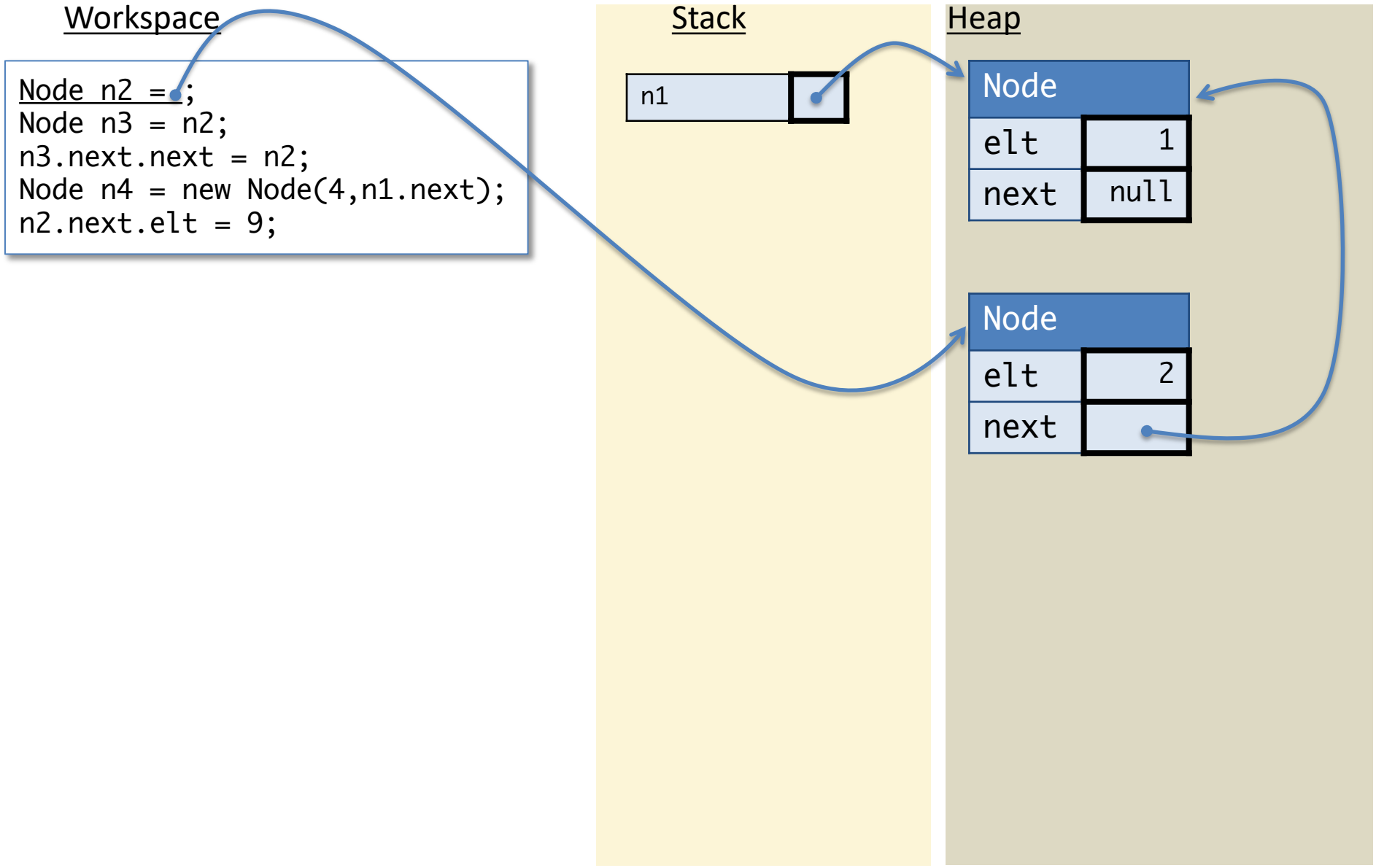


Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

Workspace

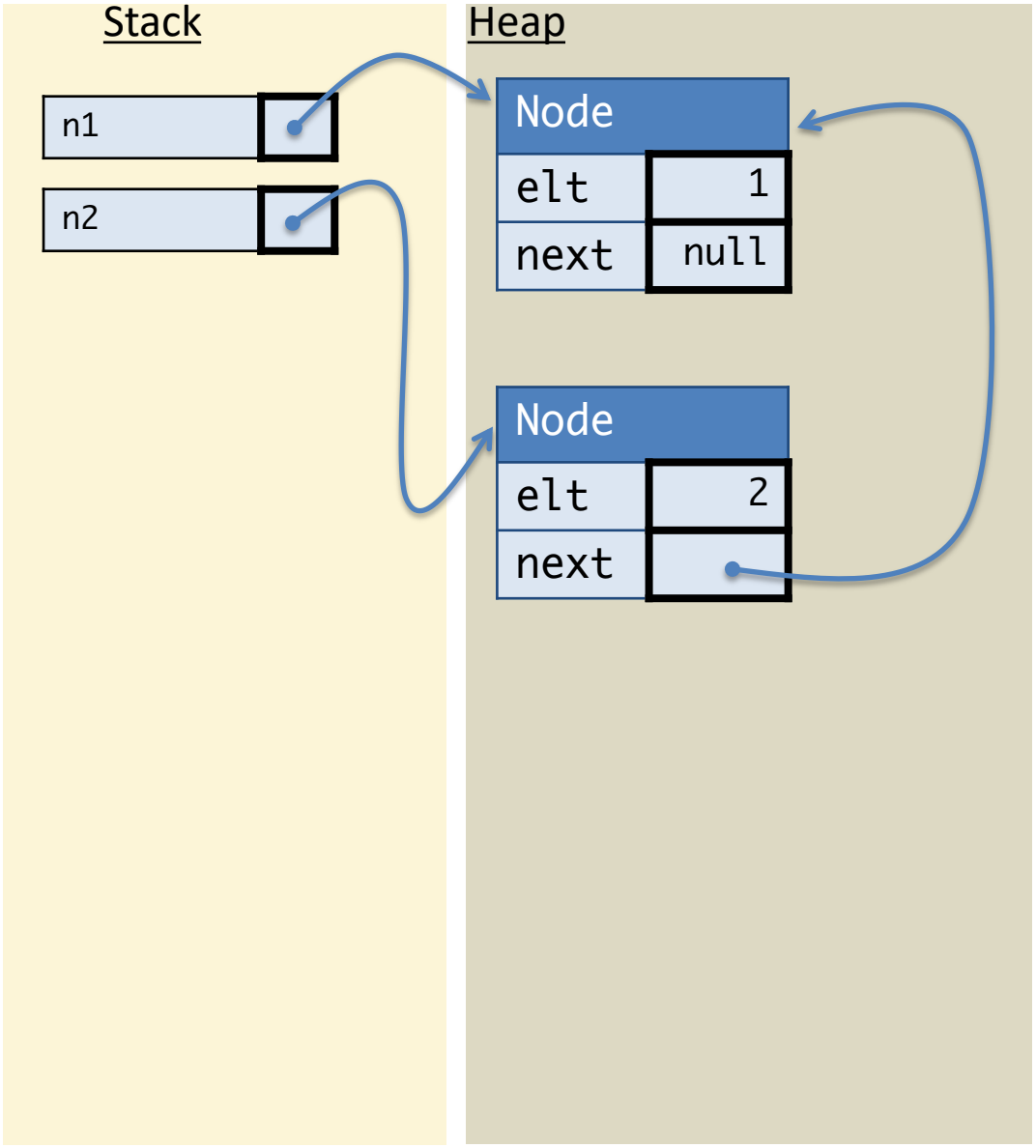
```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```




Workspace
Node n2 =;
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
Stack
n1
Heap
Node
elt
1
next
null
Node
elt
2
next

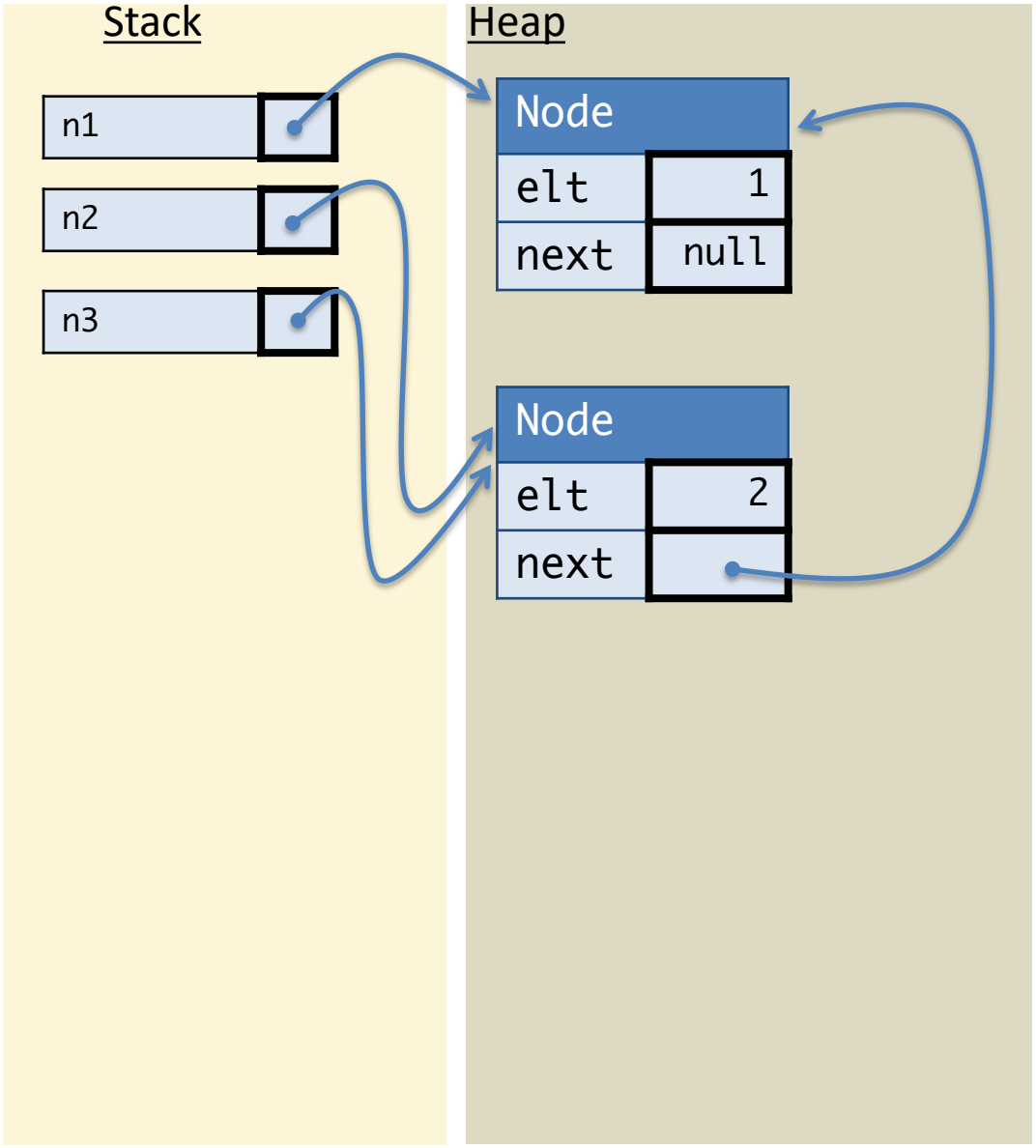
Workspace

```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



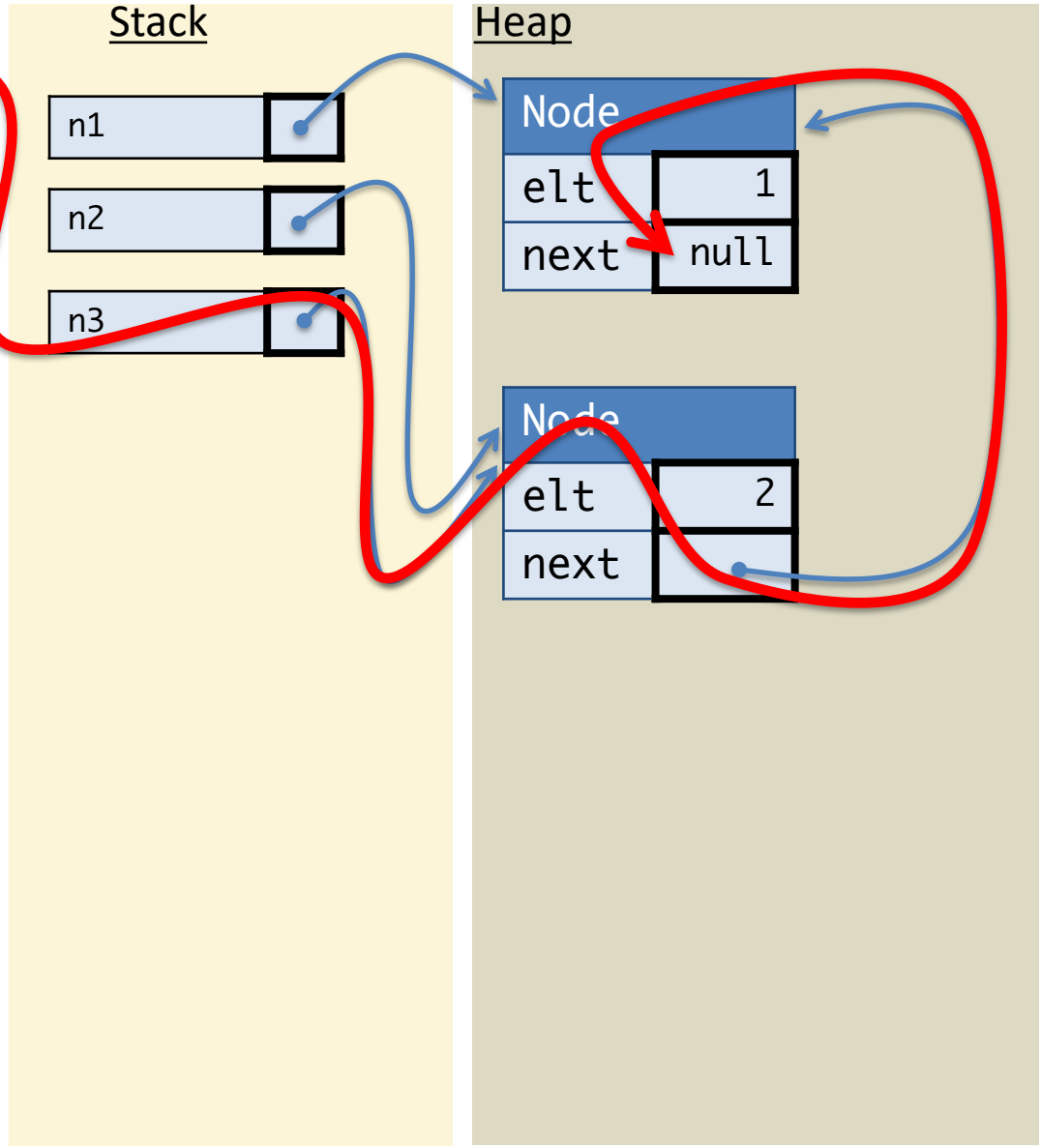
Workspace

```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



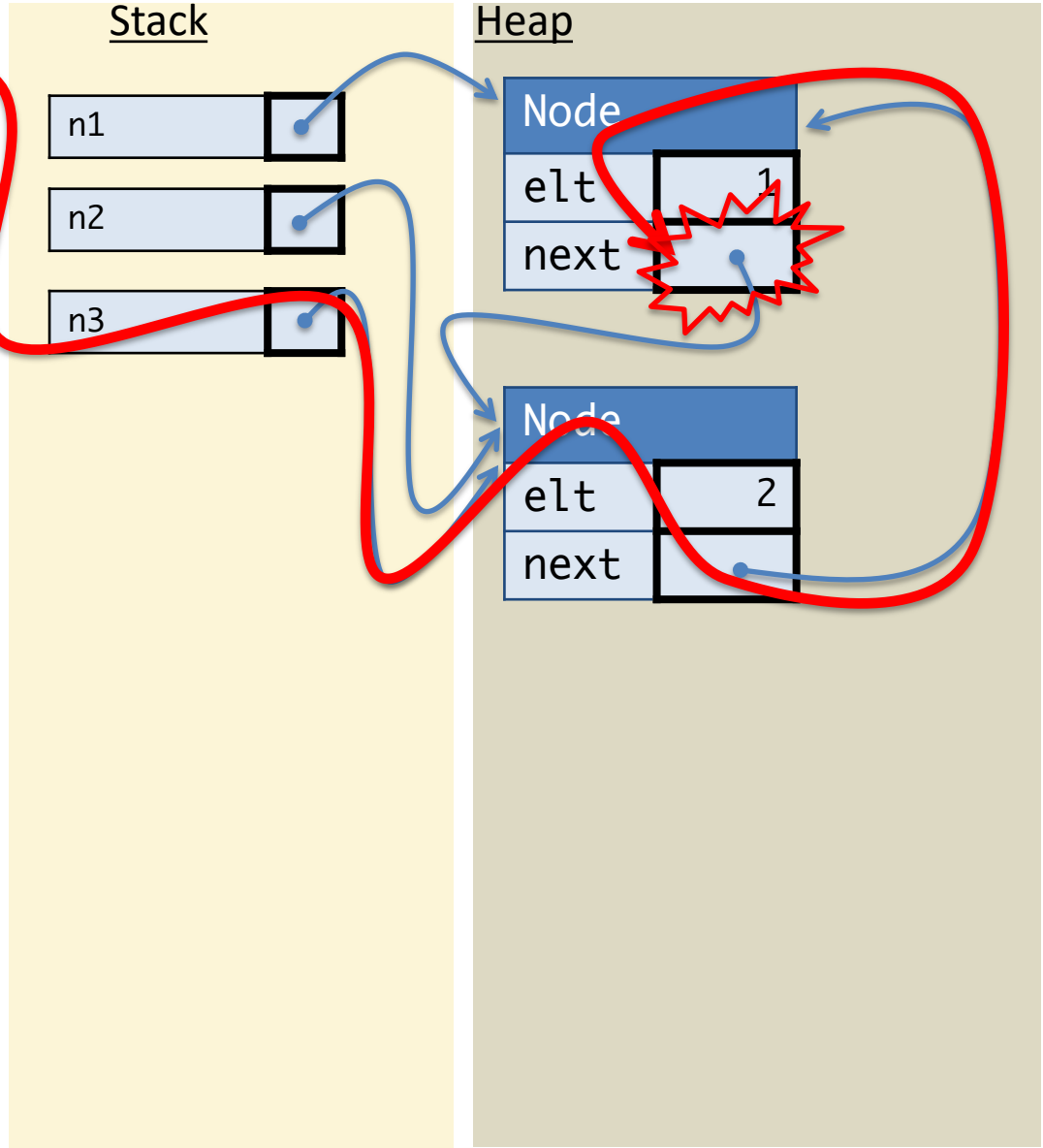
Workspace

```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



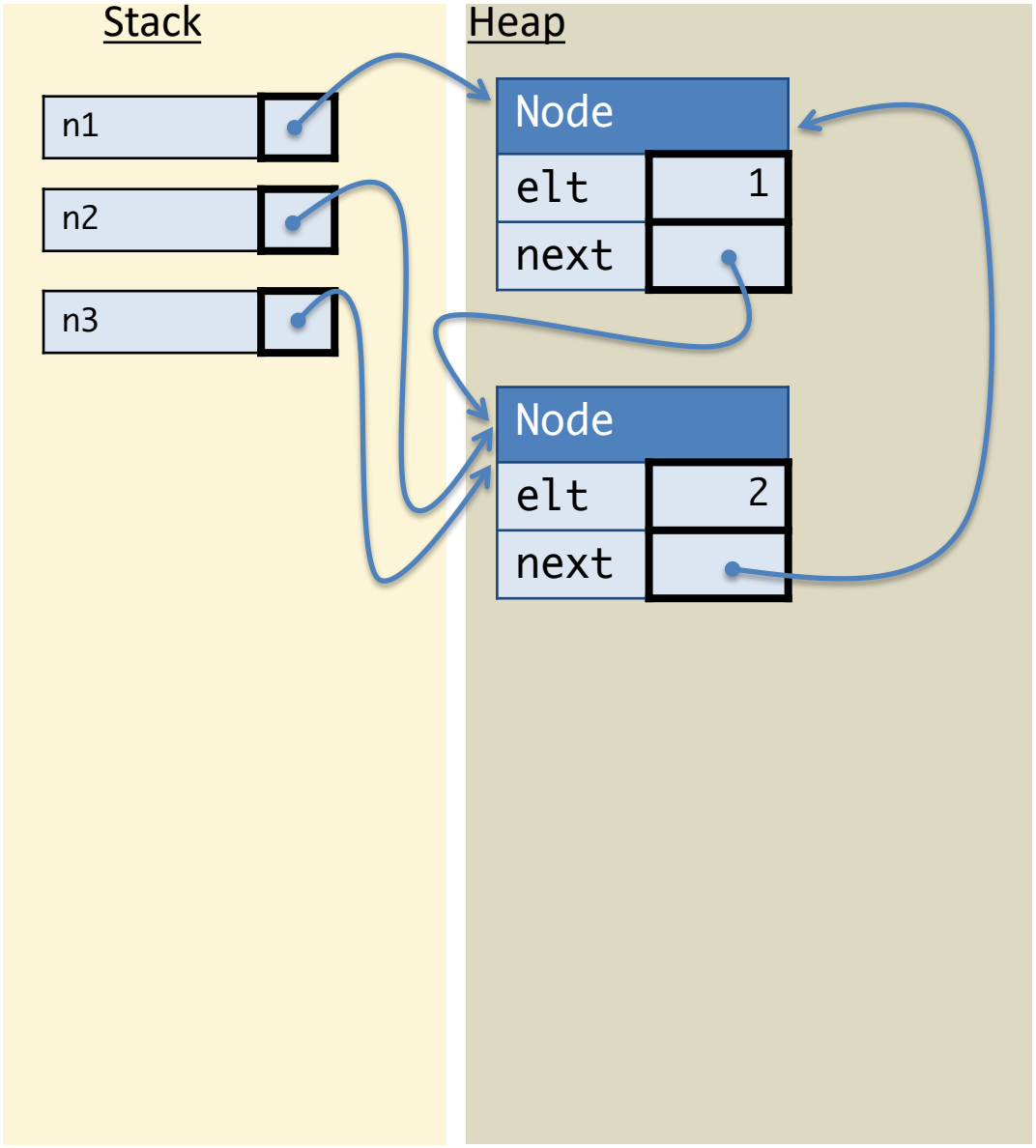
Workspace

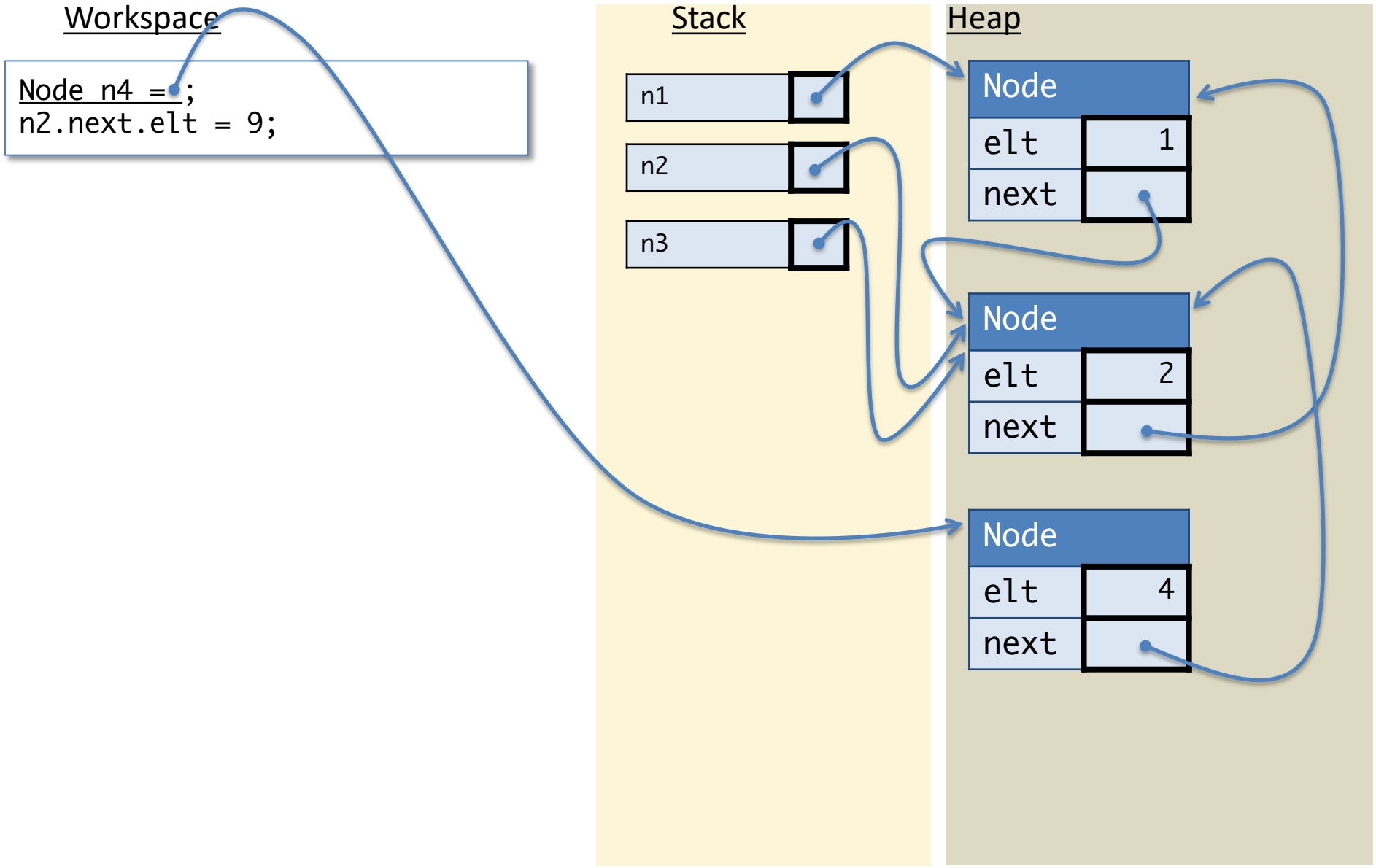
```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



Workspace

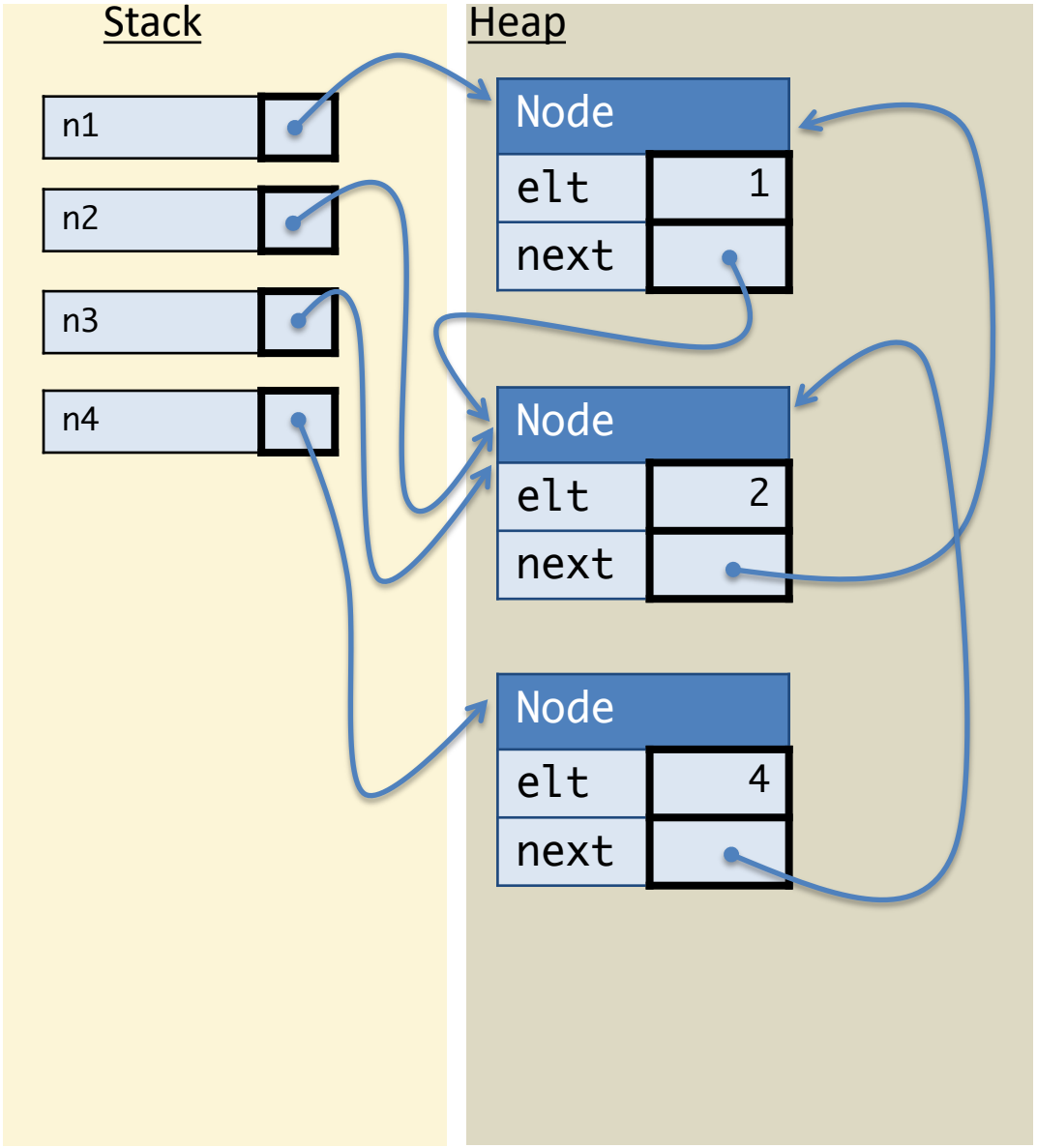
```
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

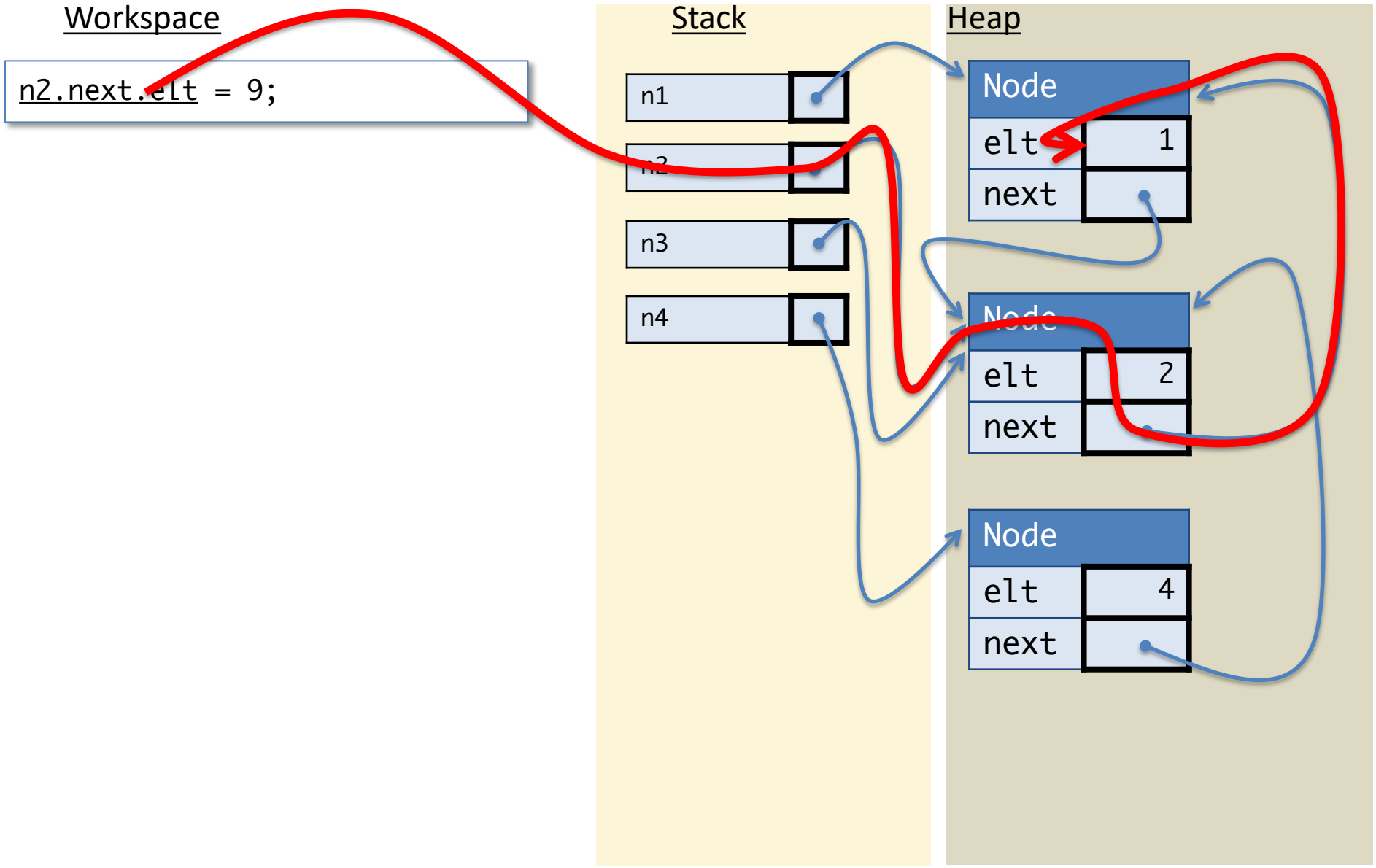



Workspace
Node n4 =;
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

Workspace

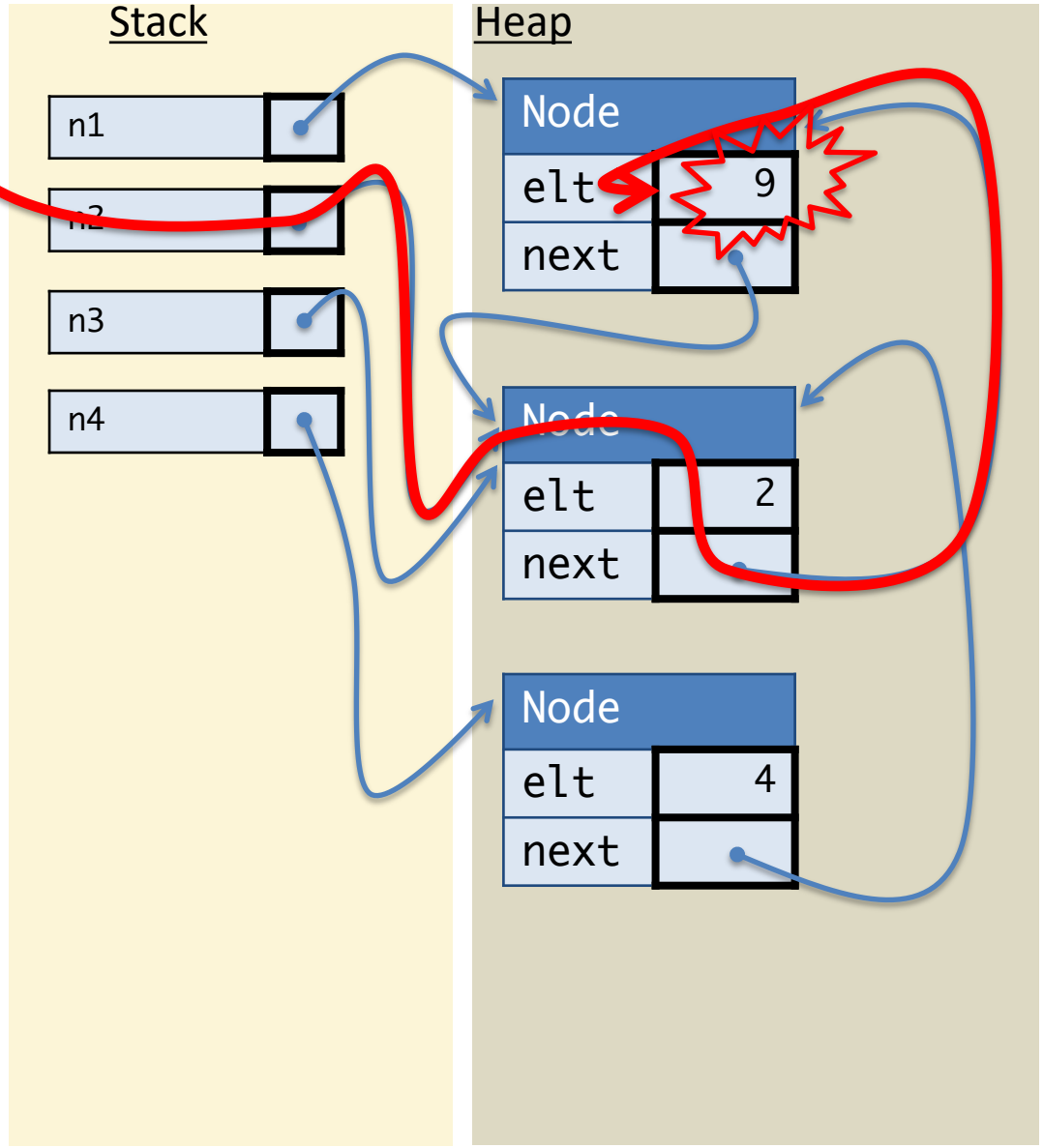
```
n2.next.elt = 9;
```

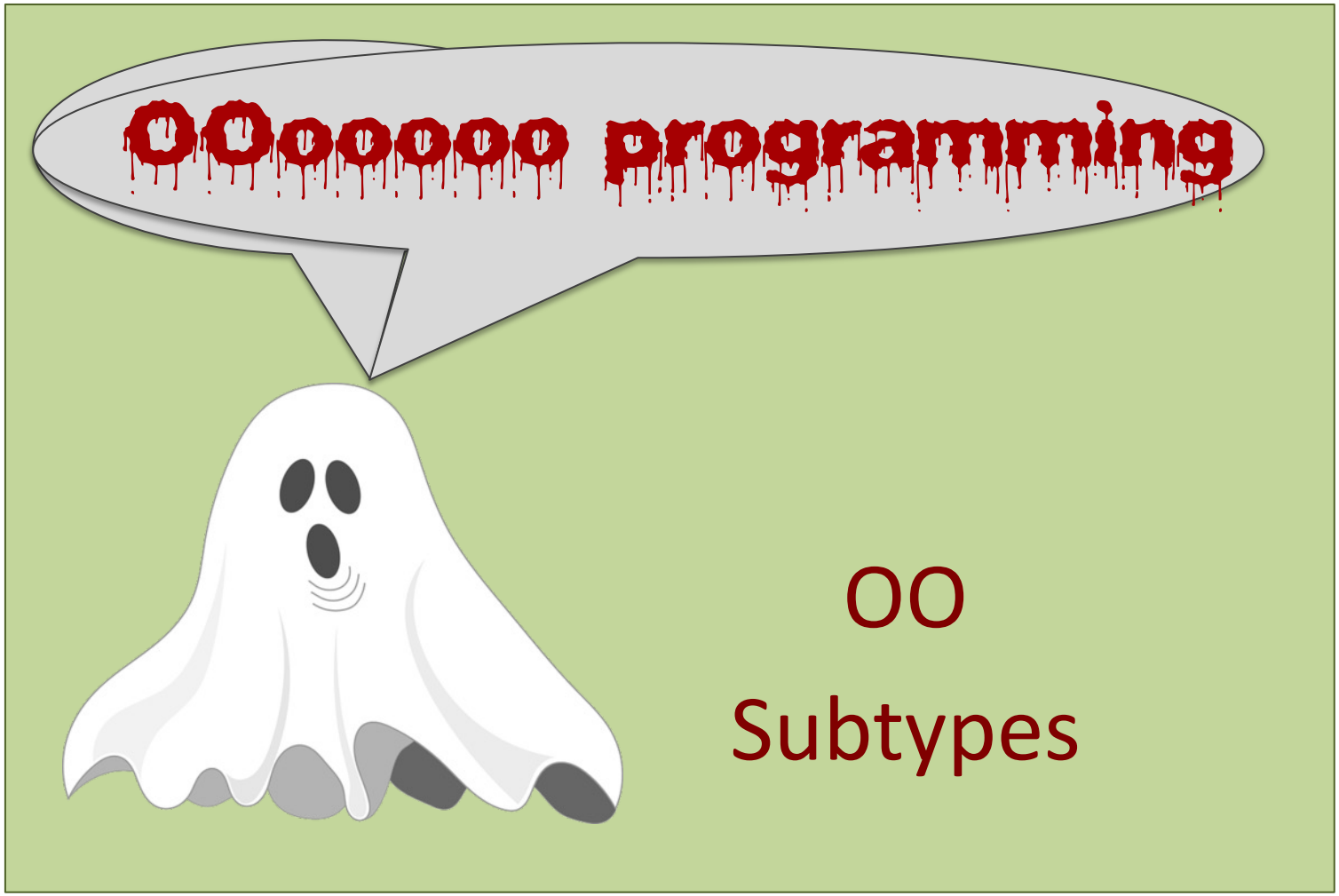



Workspace
n2.next.elt = 9;
Stack
n1
n2
n3
n4
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

Workspace

```
n2.next.elt = 9;
```




OOoooooo programming
OO
Subtypes

Review: Static Types

- Types stop you from using values incorrectly
  - `3 + true`
  - `(new Counter()).m()`
- All *expressions* have types
  - `3 + 4` has type `int`
  - "A".`toLowerCase()` has type `String`
- How do we know if `x.m()` is correct? or `x+3`?
  - depends on the type of `x`
- Type restrictions preserve the types of variables
  - assignment "x = 3" must be to values with compatible types
  - methods "o.m(3)" must be called with compatible arguments

HOWEVER: in Java, values can have *multiple* types....

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describes a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

keyword

```
public interface Displaceable {
   int getX();
   int getY();
   void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface must provide appropriate definitions for the methods specified in the interface

```
public class Point implements Displaceable {
  private int x, y;
  public Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

interfaces implemented

methods required to satisfy contract

Another implementation

```
public class Circle implements Displaceable {
  private Point center;
  private int radius;
  public Circle(int x, int y, int initRadius) {
    Point center = new Point(x, y);
    radius = initRadius;
  }
  public int getX() { return center.getX(); }
  public int getY() { return center.getY(); }
  public void move(int dx, int dy) {
    center.move(dx, dy);
  }
}
```

Objects with different local state can satisfy the same interface

Implementing multiple interfaces

```
public interface Area {
    public double getArea();
}
```

```
public class Circle implements Displaceable, Area {
  private Point center;
  private int radius;
  // constructor
  // implementation of Displaceable methods

  // new method
  public double getArea() {
     return Math.pi * radius * radius;
  }

}
```

Classes can implement multiple interfaces by including *all* of the required methods

24: Assume Circle implements the Displaceable interface. The following snippet of code typechecks:

0

```
// in class C
public static void moveItALot (Displaceable s) {
    … //omitted
}

… // elsewhere
Circle c = new Circle(new Point(10,10),10);
C.moveItALot(c);
```

True

0%

False

0%

Assume Circle implements the Displaceable interface.

The following snippet of code typechecks:

```
// in class C
public static void moveItALot (Displaceable s) {
    … //omitted
}

… // elsewhere
Circle c = new Circle(new Point(10,10),10);
C.moveItAlot(c);
```

1. True
2. False

Answer: True

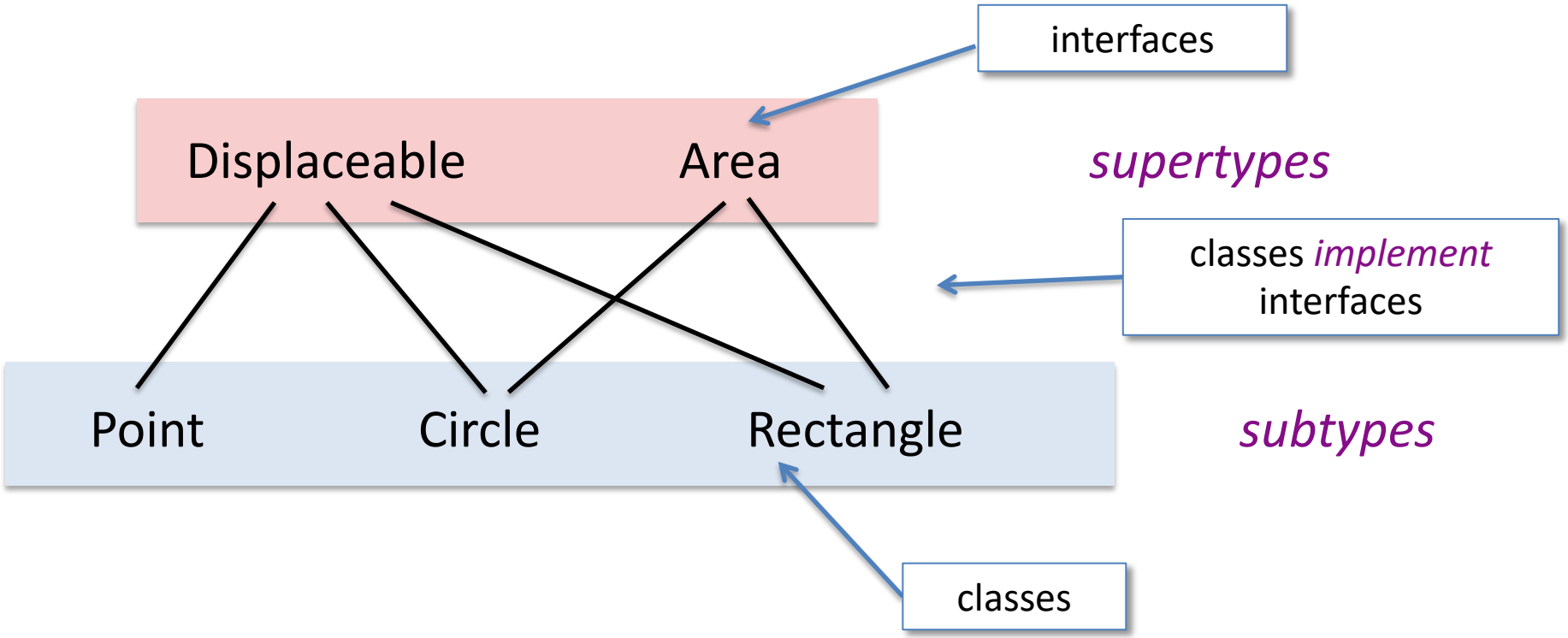
Subtyping

Definition: Type A can be declared to be a *subtype* of type B if values of type A can do anything that values of type B can do. Type B is called a *supertype* of A.

Example: A class that implements an interface declares a subtyping relationship

Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces



Types can have many *different* supertypes / subtypes

Subtype Polymorphism*

- Main idea:

> Anywhere an object of type A is needed, an object that actually belongs to a subtype of A can be provided.

```
// in class C
public static void leapIt(Displaceable c) {
    c.move(1000,1000);
 }
// somewhere else
C.leapIt(new Circle (p, 10));
```

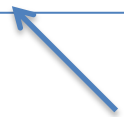
- If B is a subtype of A, it provides all of A's (public) methods
- The behavior of a nonstatic method (like move) depends on B's implementation

**polymorphism = "many shapes"*

Subtyping and Variables

- A a *variable* declared with type A can store any *object* that is a subtype of A

```
Displaceable a = new Circle(new Point(2,3), 1);
```



supertype of Circle

subtype of Displaceable

- Methods with *parameters* of type A must be called with *arguments* that are subtypes of A