Programming Languages and Techniques (CIS120)

Lecture 24

March 15, 2013

Java ASM

Subtyping and Extension

Announcements

- HW 07 due Monday at midnight

- Exam II, in class, two weeks from today

The Java Abstract Stack Machine

Objects Arrays and Static Methods

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

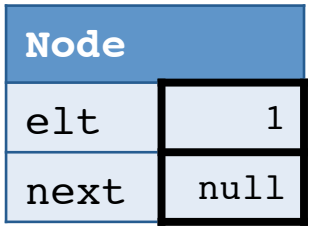
- Differences:
  - Everything, including stack slots, is mutable by default
  - Heap objects store *dynamic class information*

Heap Values

Objects

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

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

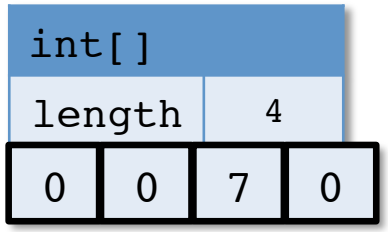


fields may or may not be mutable

Arrays

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

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



length *never* mutable
elements *always* mutable

Resizable Arrays

```
public class ResArray {

  /** Constructor, takes no arguments. */
  public ResArray() { … }

  /** Access the array at position i. If position i has not yet
   * been initialized, return 0.
   */
  public int get(int i) { … }

  /** Modify the array at position i to contain the value v. */
  public void set(int i, int v) { … }

  /** Return the extent of the array. */
  public int getExtent() { … }

}
```

Object Invariant: extent is always 1 past the last nonzero value in data
(or 0 if the array is all zeros)

ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```

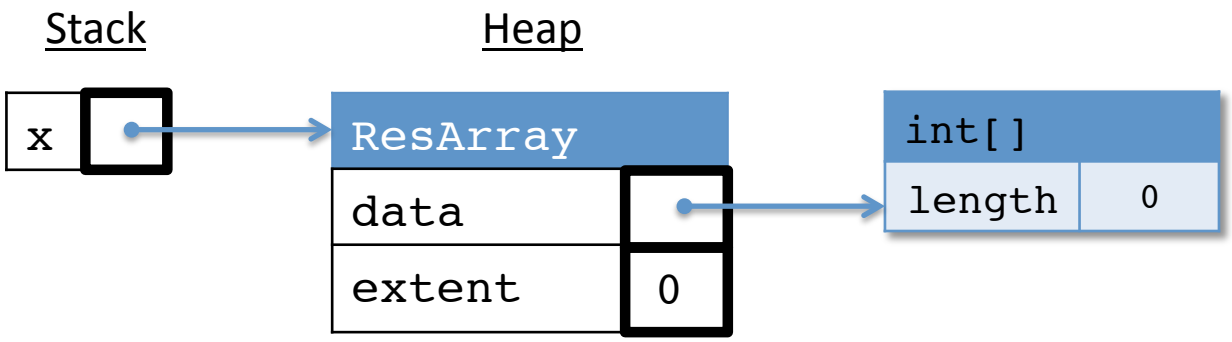
Stack

Heap

ResArray ASM

Workspace

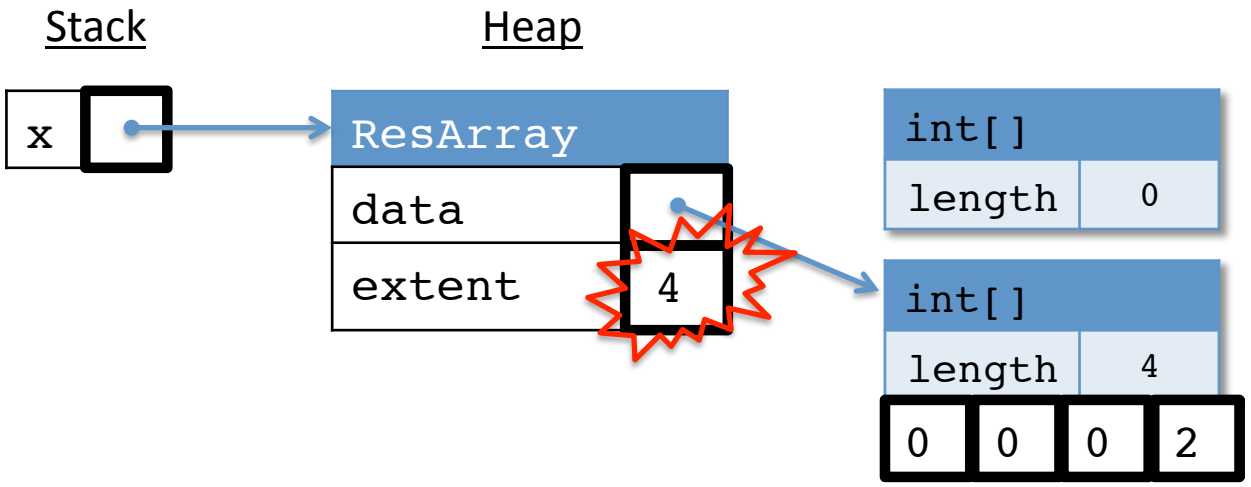
```
~~ResArray x = new ResArray();~~
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

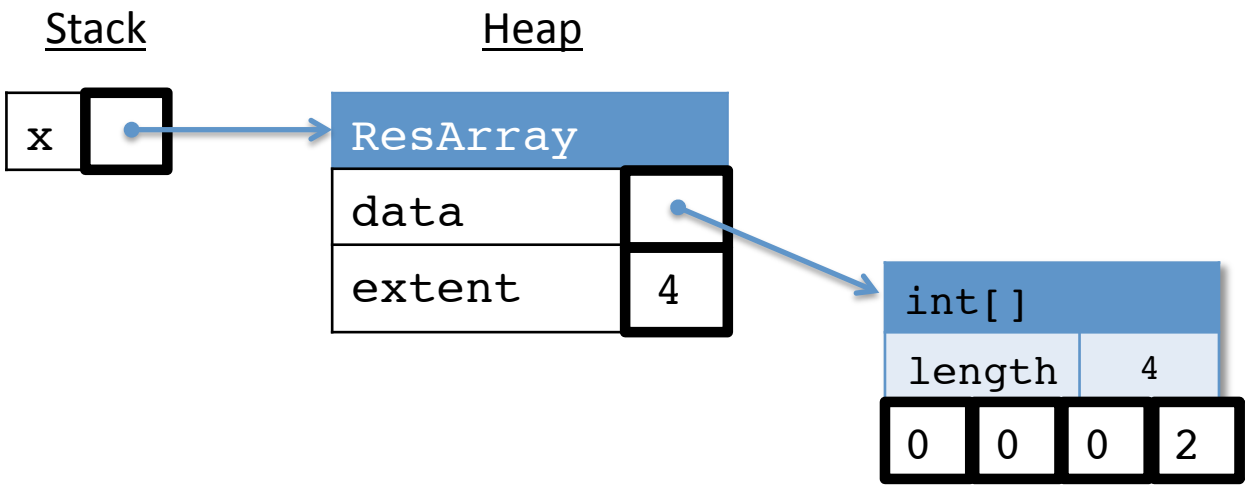
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

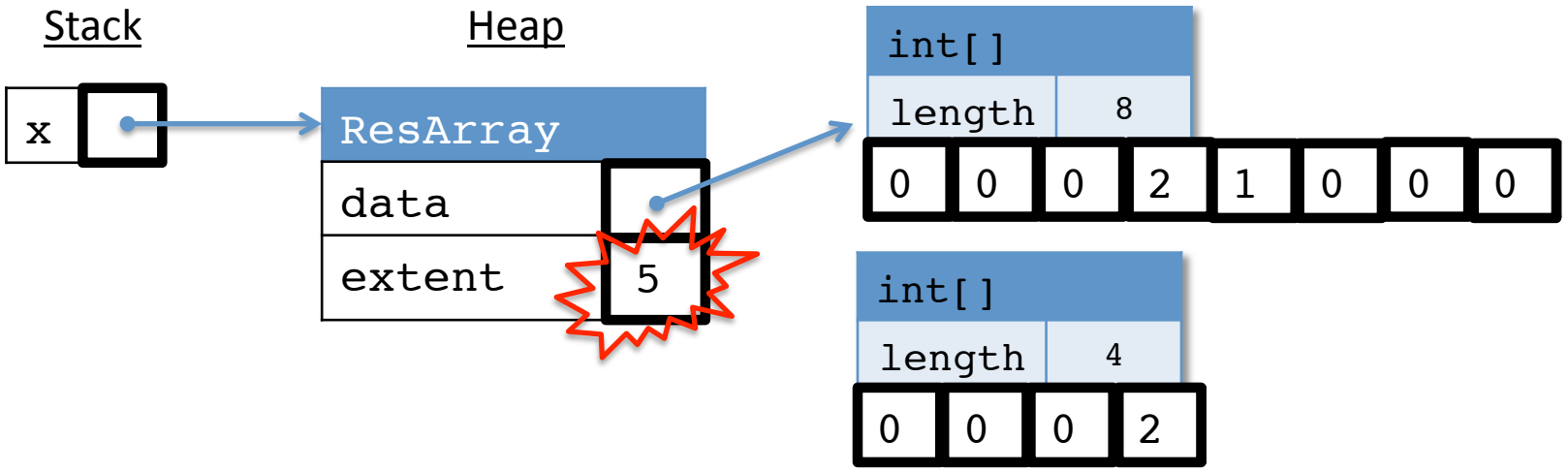
```
~~ResArray x = new ResArray();~~
~~x.set(3,2);~~
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

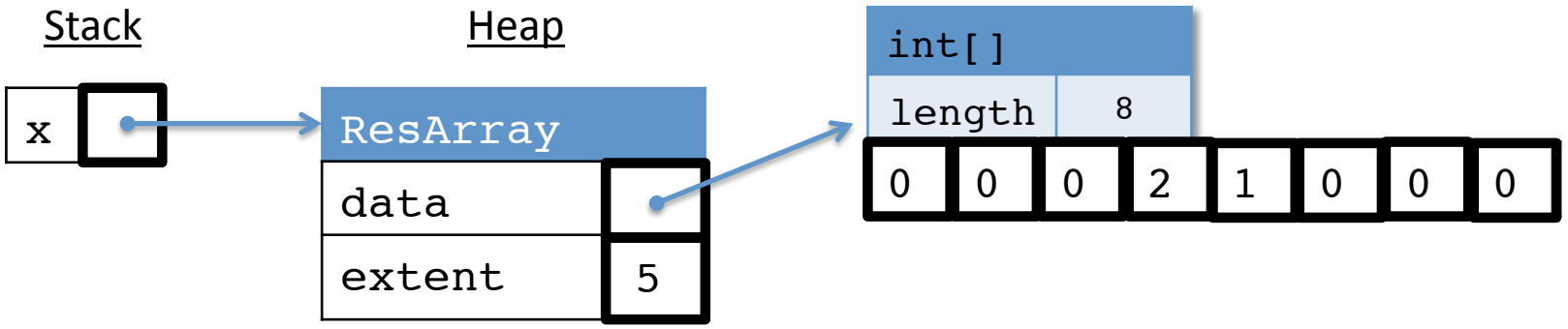
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

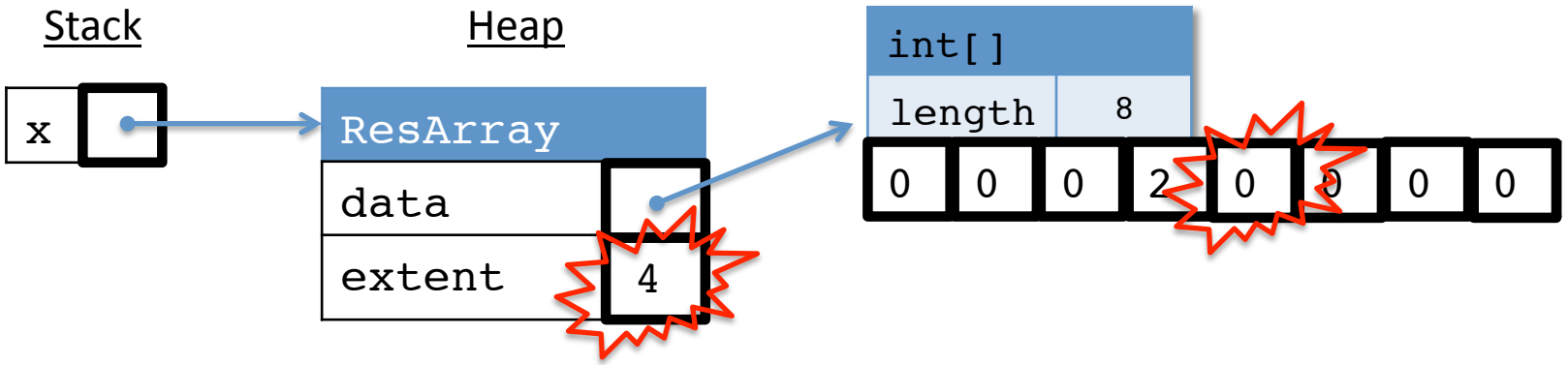
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



Resizable Arrays

Object Invariant: extent is always 1 past the last nonzero value in data
(or 0 if the array is all zeros)

```
public class ResArray {

  /** Constructor, takes no arguments. */
  public ResArray() { … }

  /** Access the array at position i. If position i has not yet
   * been initialized, return 0.
   */
  public int get(int i) { … }

  /** Modify the array at position i to contain the value v. */
  public void set(int i, int v) { … }

  /** Return the extent of the array. */
  public int getExtent() { … }

  /** The smallest prefix of the ResArray
   * that contains all of the nonzero values as a normal array.
   */
  public int[] values() { … }
}
```

Values Method

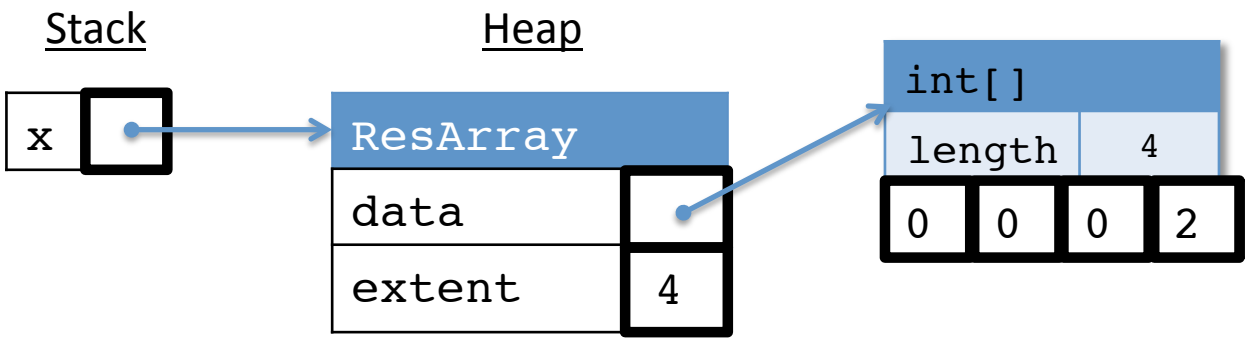
```
public int[] values() {
    int[] values = new int[extent];
    for(int i=0; i<extent; i++) {
        values[i] = data[i];
    }
    return values;
}
```

```
public int[] values() {
    return data;
}
```

ResArray ASM

Workspace

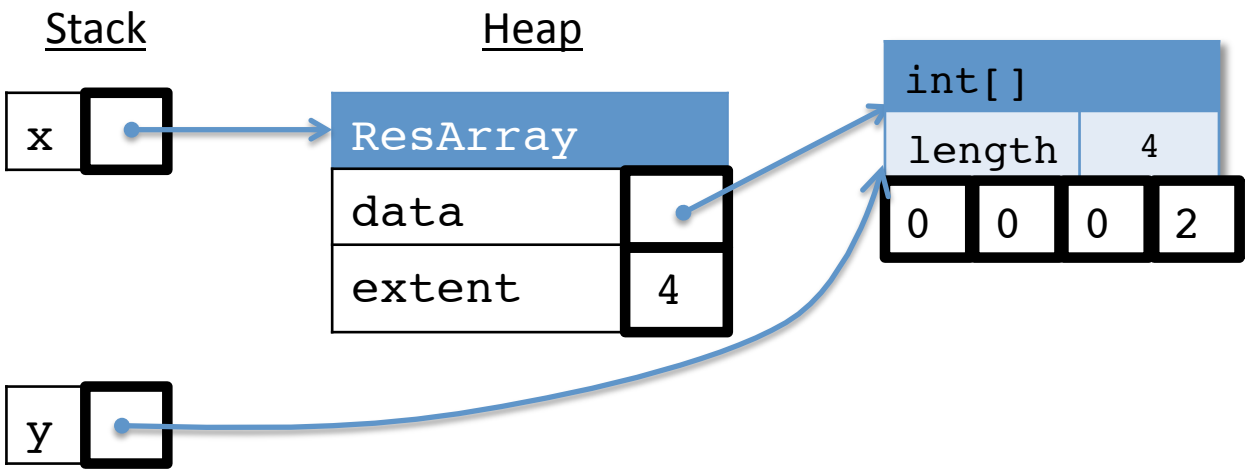
```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



ResArray ASM

Workspace

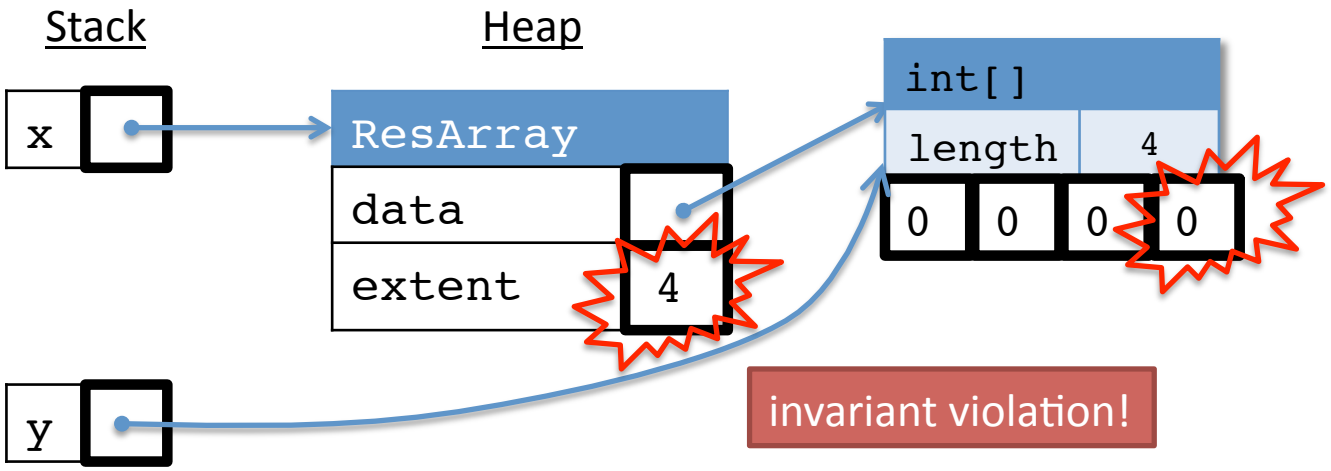
```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



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.

Object Aliasing example

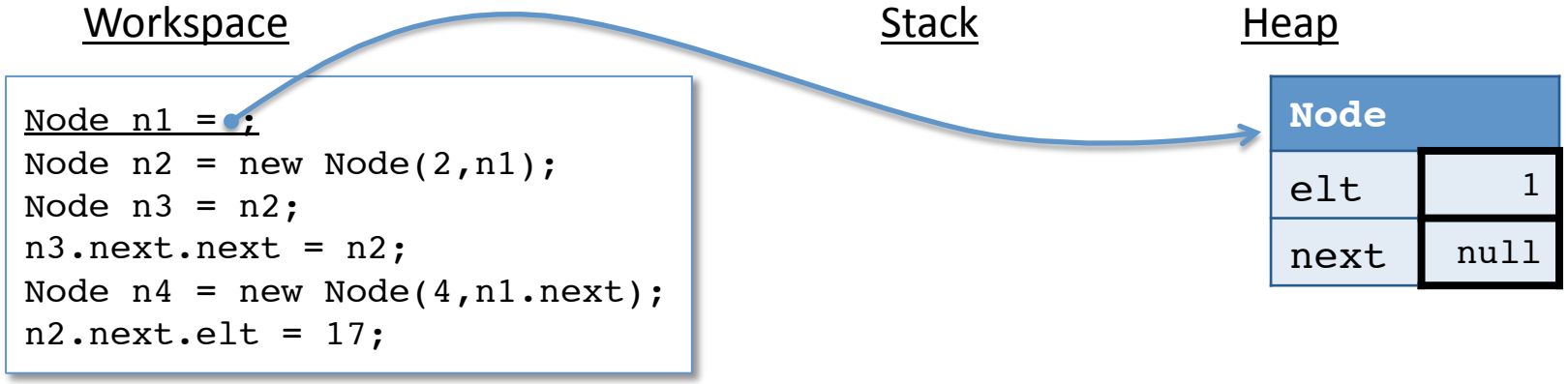
```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}

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 = 17;
    System.out.println(n1.next);
}
```

Constructing an Object

Workspace | Stack | Heap

```
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 = 17;
```

Constructing an Object



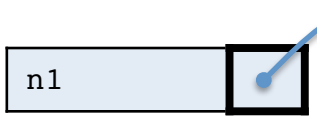
Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

Constructing an Object

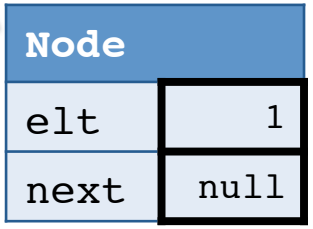
Workspace

```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```

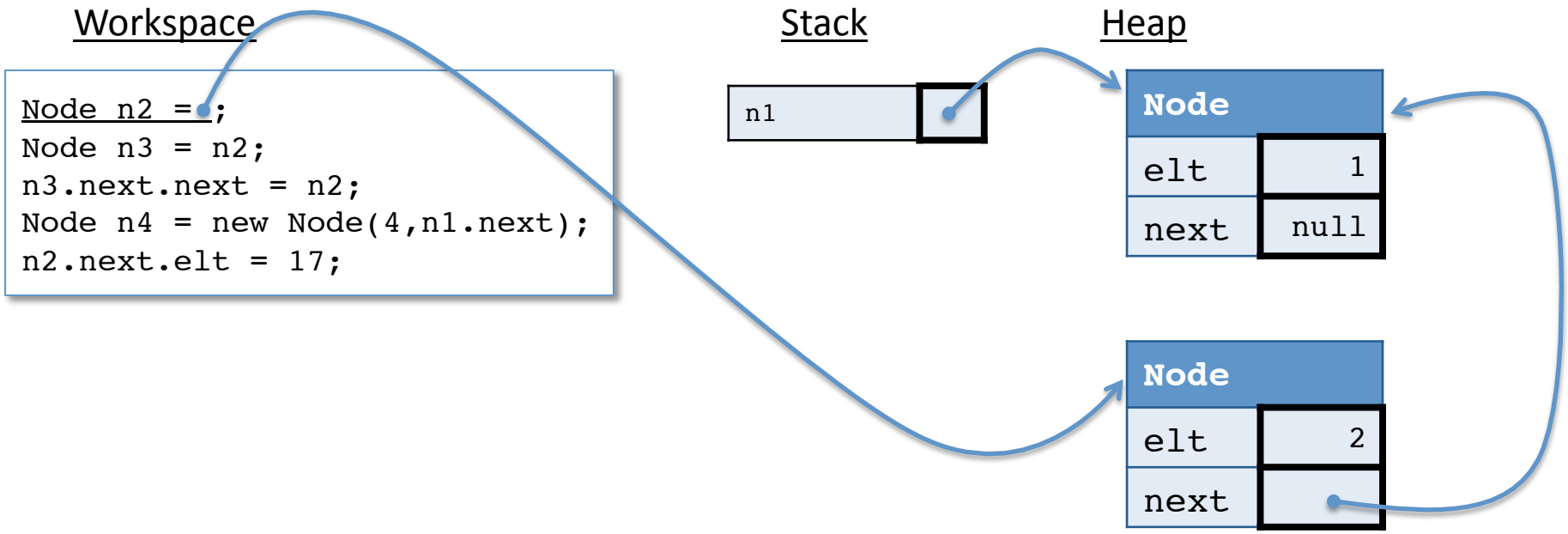
Stack



Heap



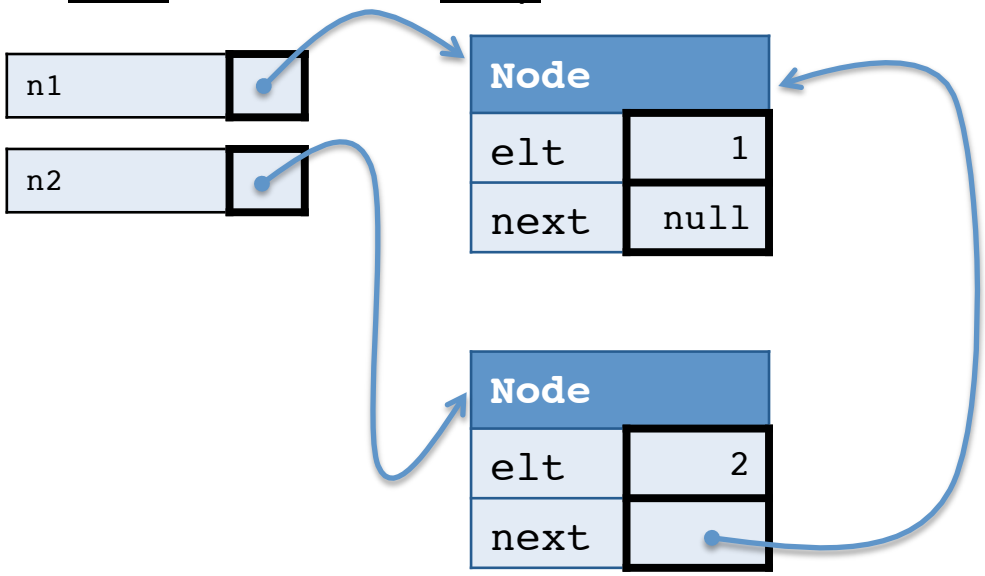
Constructing an Object



Constructing an Object

Workspace

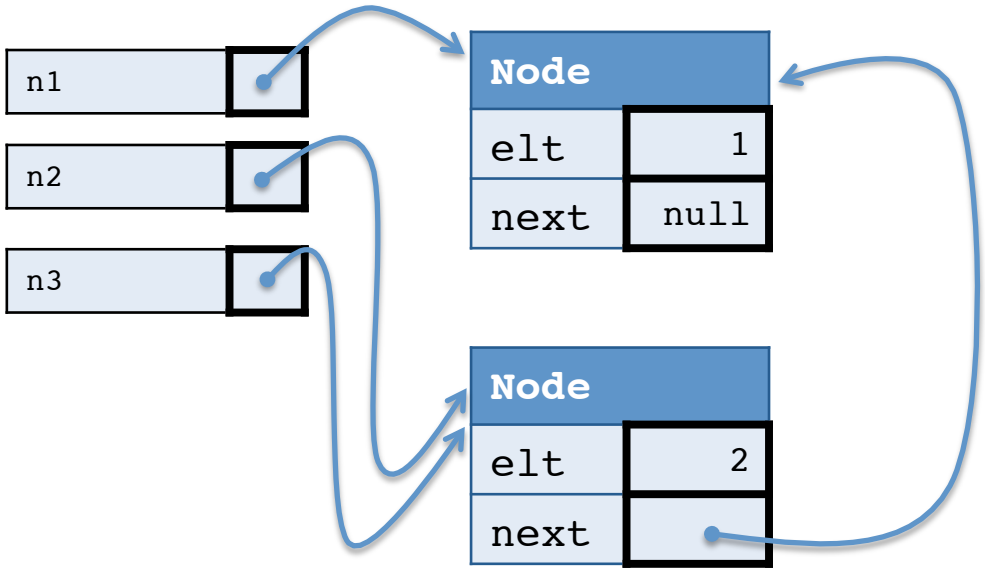
```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



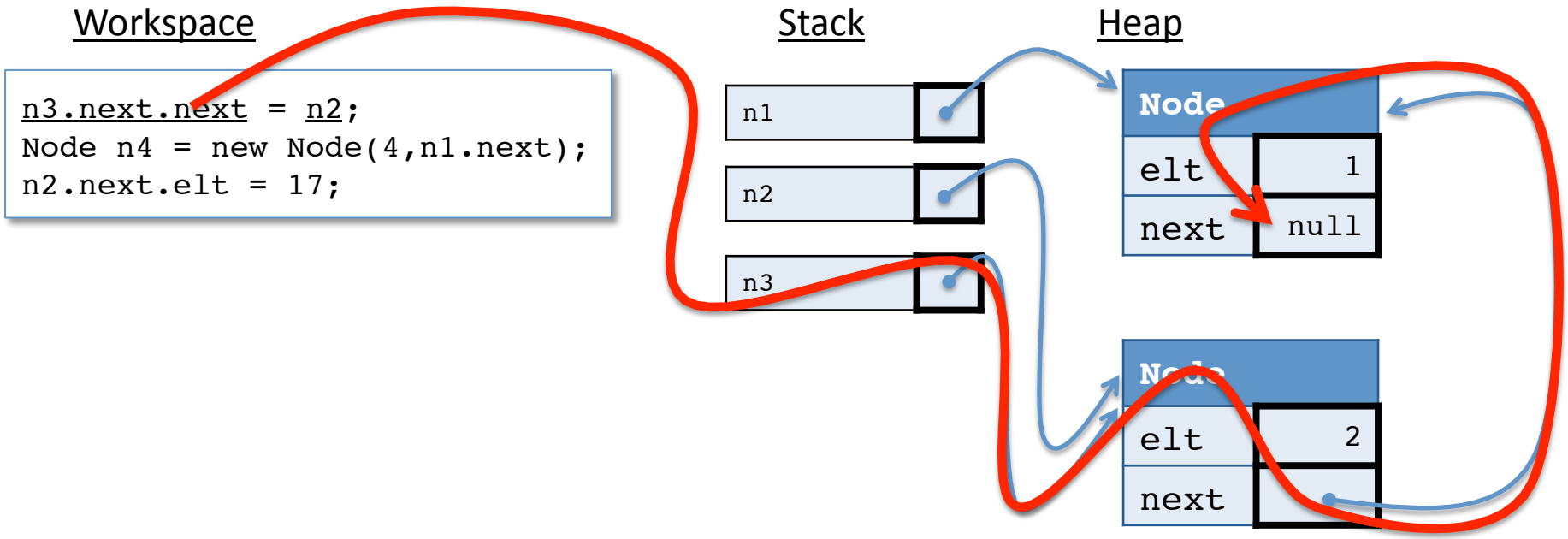
Constructing an Object

Workspace

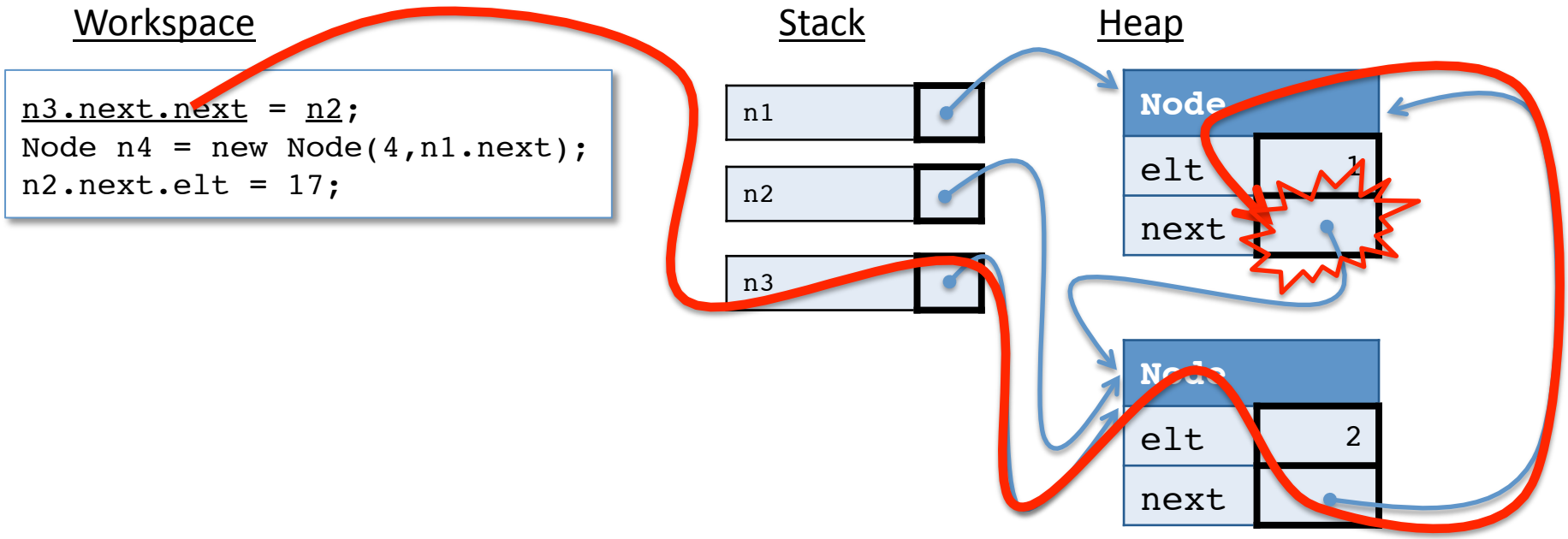
```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



Constructing an Object



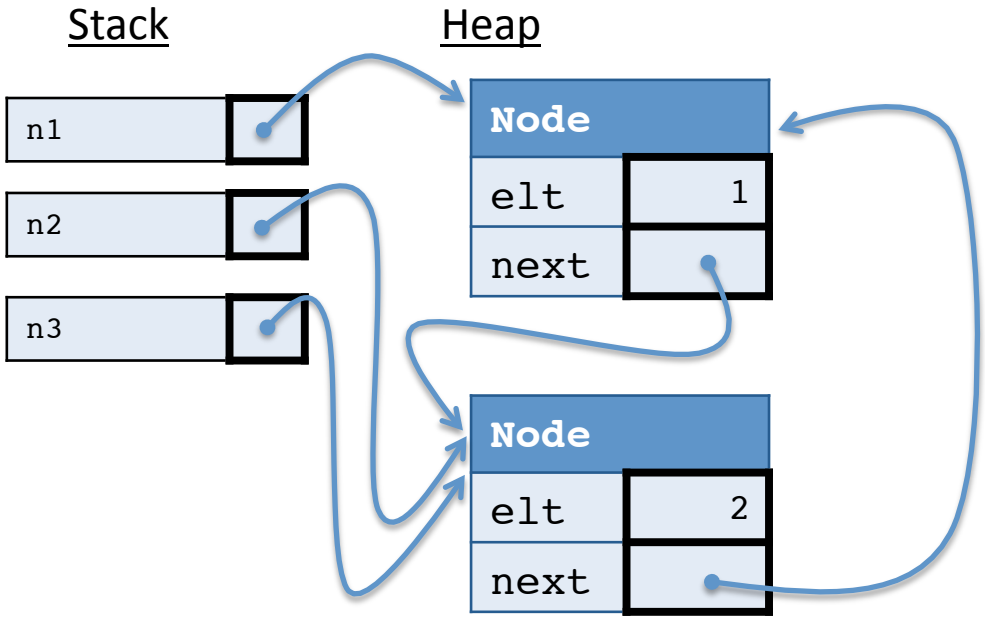
Constructing an Object



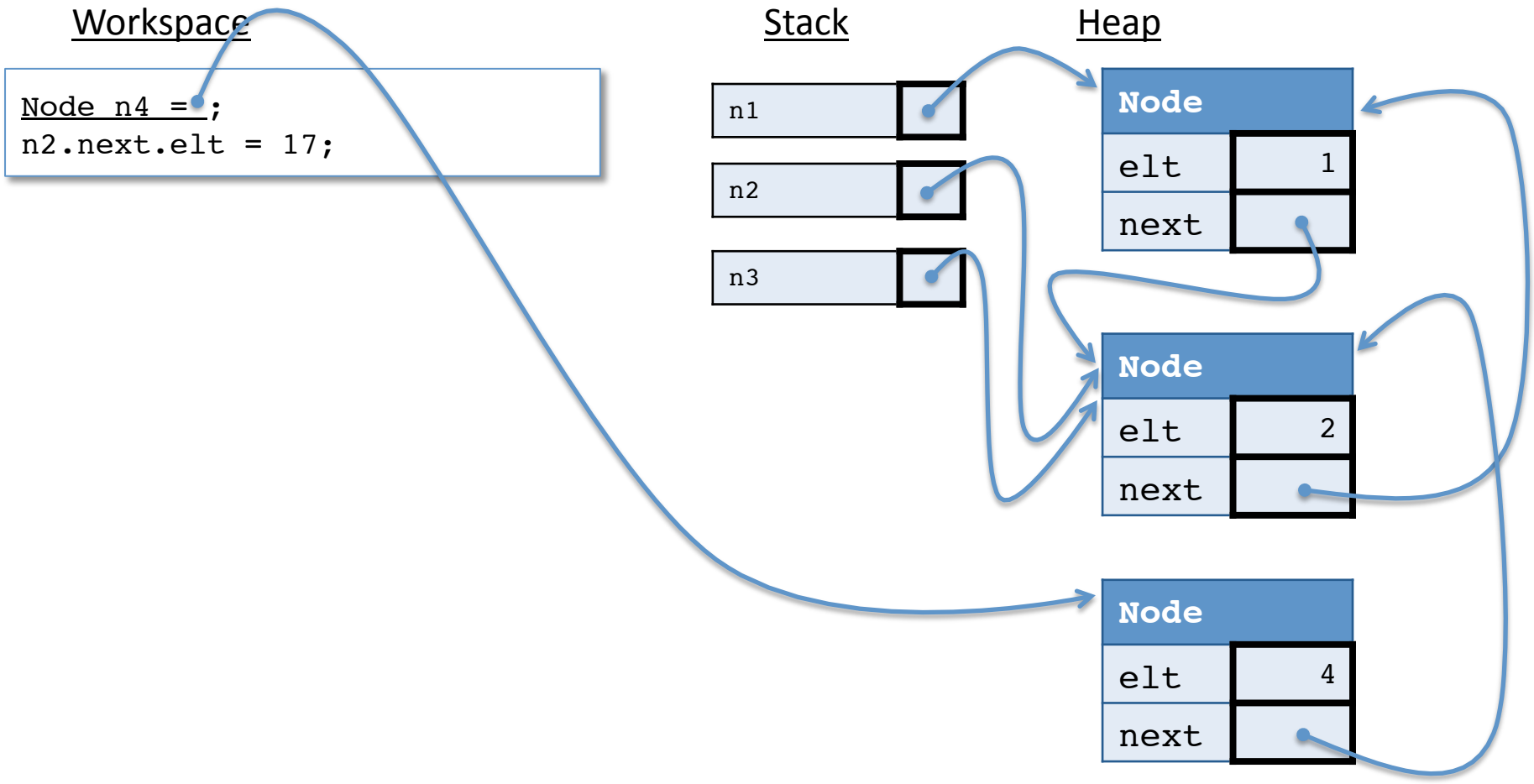
Constructing an Object

Workspace

```
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



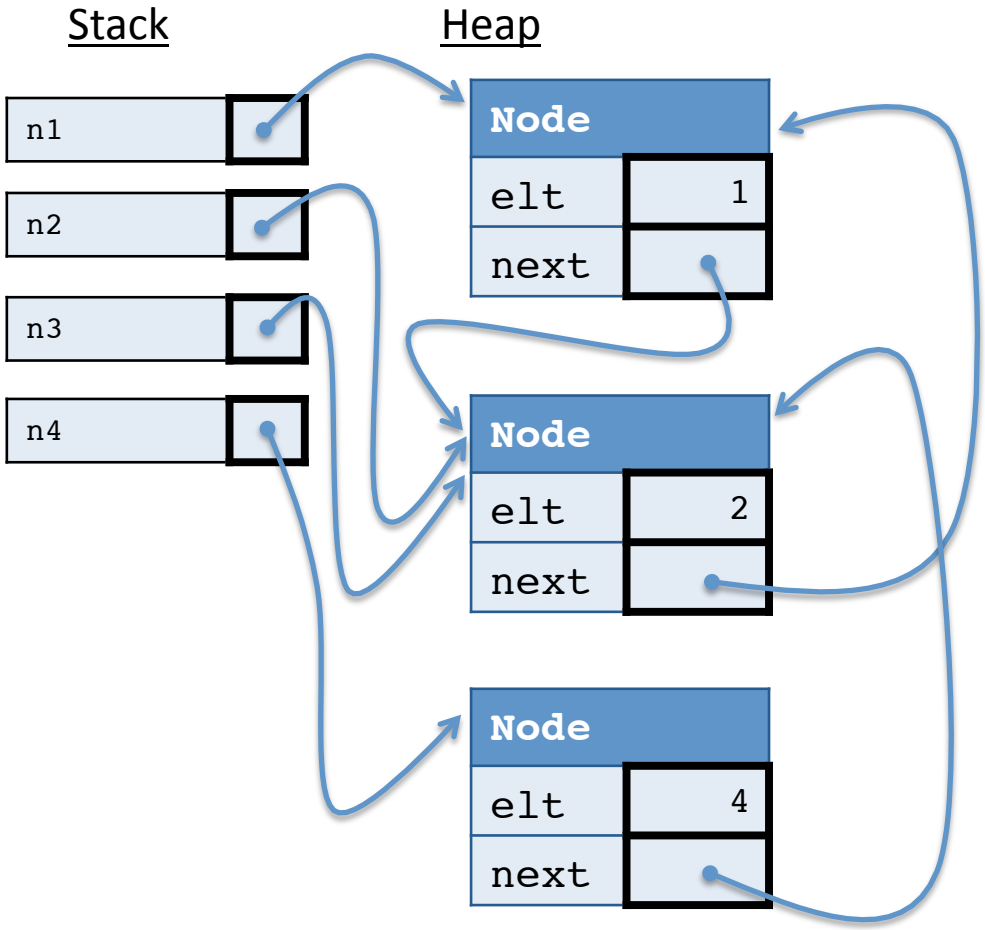
Constructing an Object



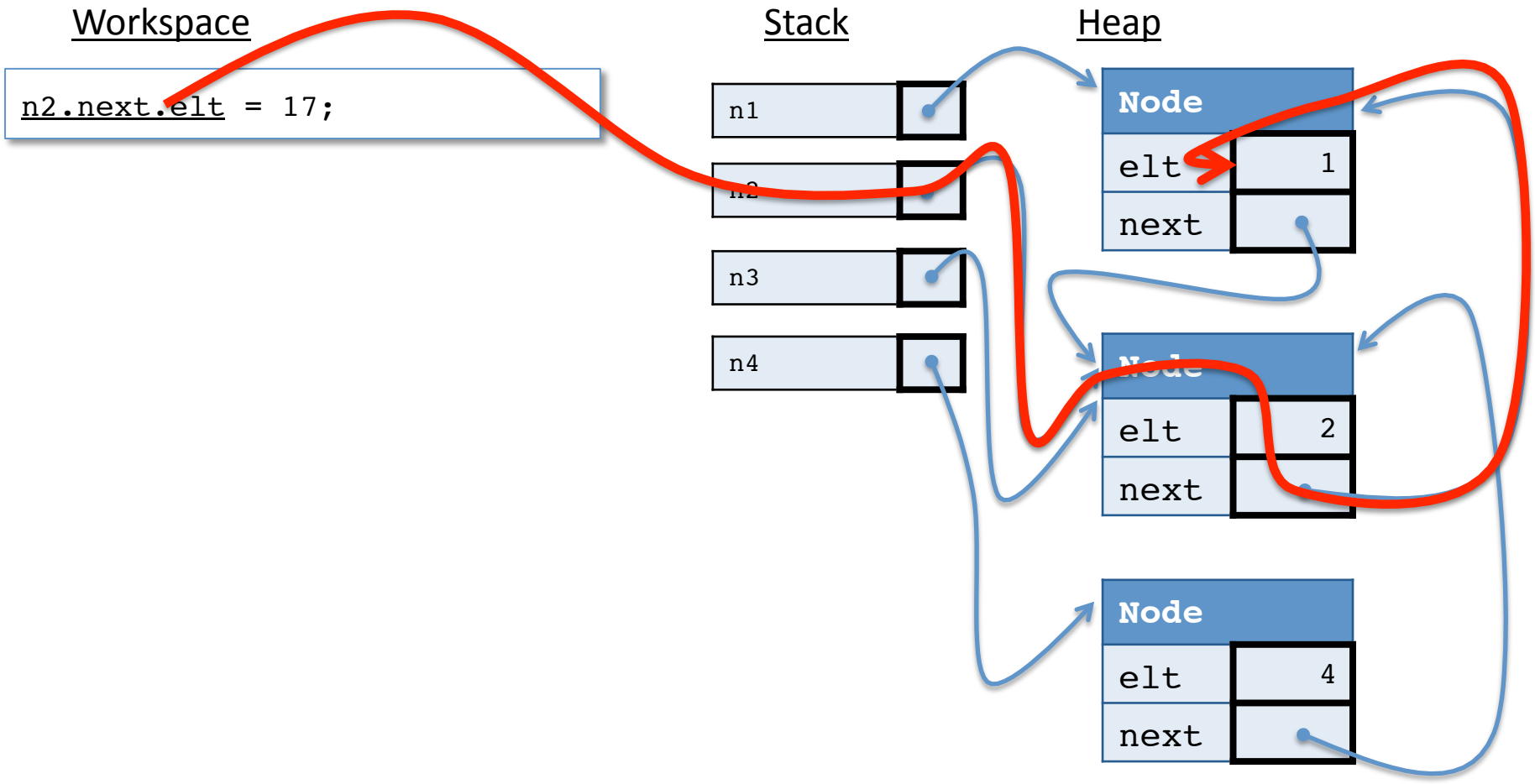
Constructing an Object

Workspace

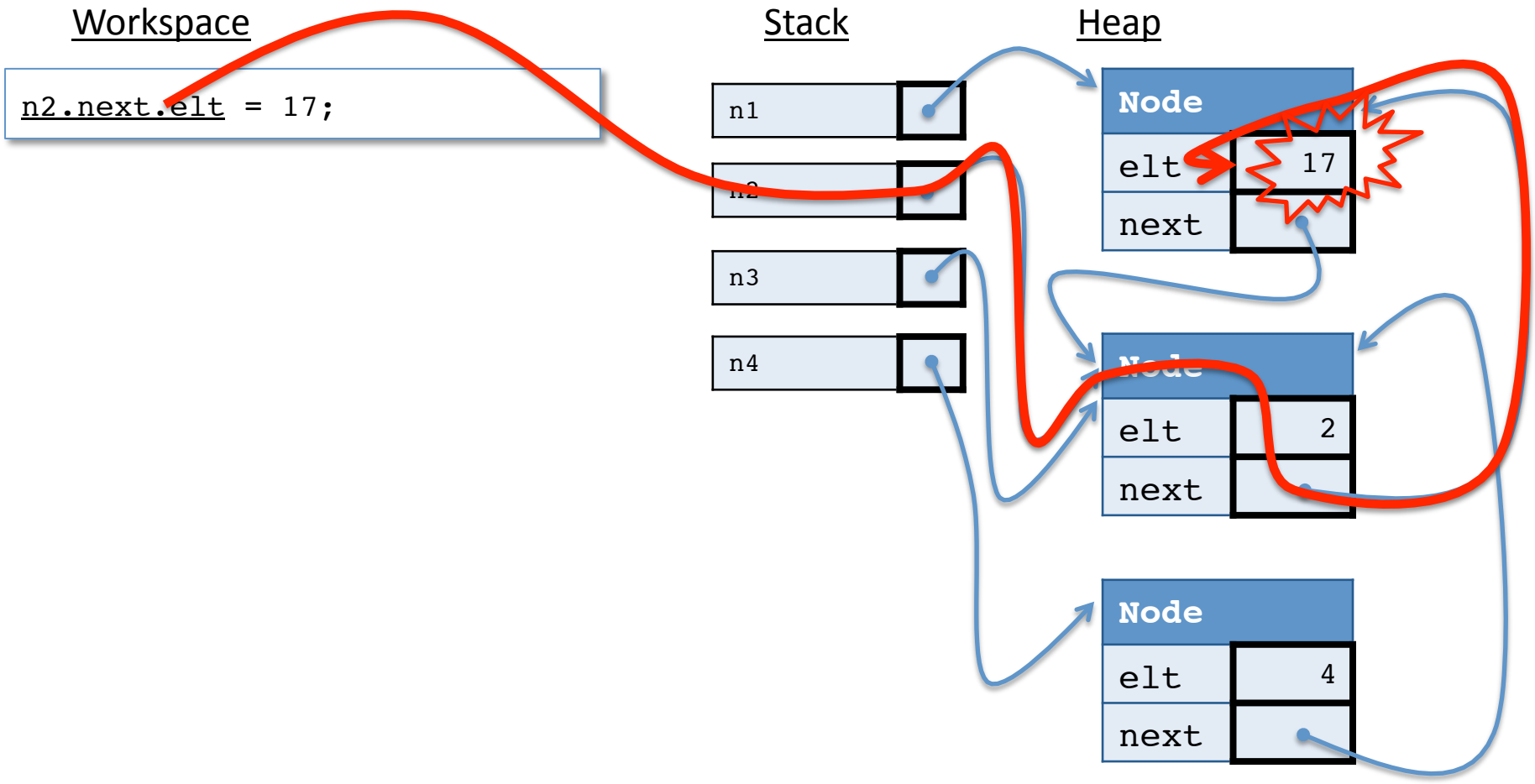
```
n2.next.elt = 17;
```



Constructing an Object



Constructing an Object



Types and Subtyping

Why Static Types?

- Types stop you from using values incorrectly
  - `3.m()`
  - `if (3) { return 1; } else { return 2; }`
  - `3 + true`
  - `(new Counter()).m()`
- All *expressions* have types
  - `3 + 4` has type `int`
  - `"A".toLowerCase()` has type `String`
  - `new ResArray()` has type `ResArray`
- How do we know if `x.m()` is correct? or `x+3`?
  - depends on the type of `x`
  - variable declarations specify types of variables
- Type restrictions preserve the types of variables
  - assignment "x = v" must be to values with compatible types
  - methods "o.m(3)" must be called with compatible argument types
- HOWEVER: in Java, values can have *multiple* types....

Subtyping

- **Definition**: Type A is a *subtype* of type B if A can do anything that B can do. Type B is called the *supertype* of A.
- **Example**: A class that implements an interface is a subtype of the interface

```
interface Area {
    public double getArea ();
}
public class Circle implements Area {
    private double r;
    private Point p;
    public Circle (double x0, double y0, double r0) {
       r = r0; p = new Point(x0,y0);
    }
    public double getArea ()  {
       return 3.14159 * r * r;
    }
    public double getRadius () { return r; }
}
```

Subtyping and Variables

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

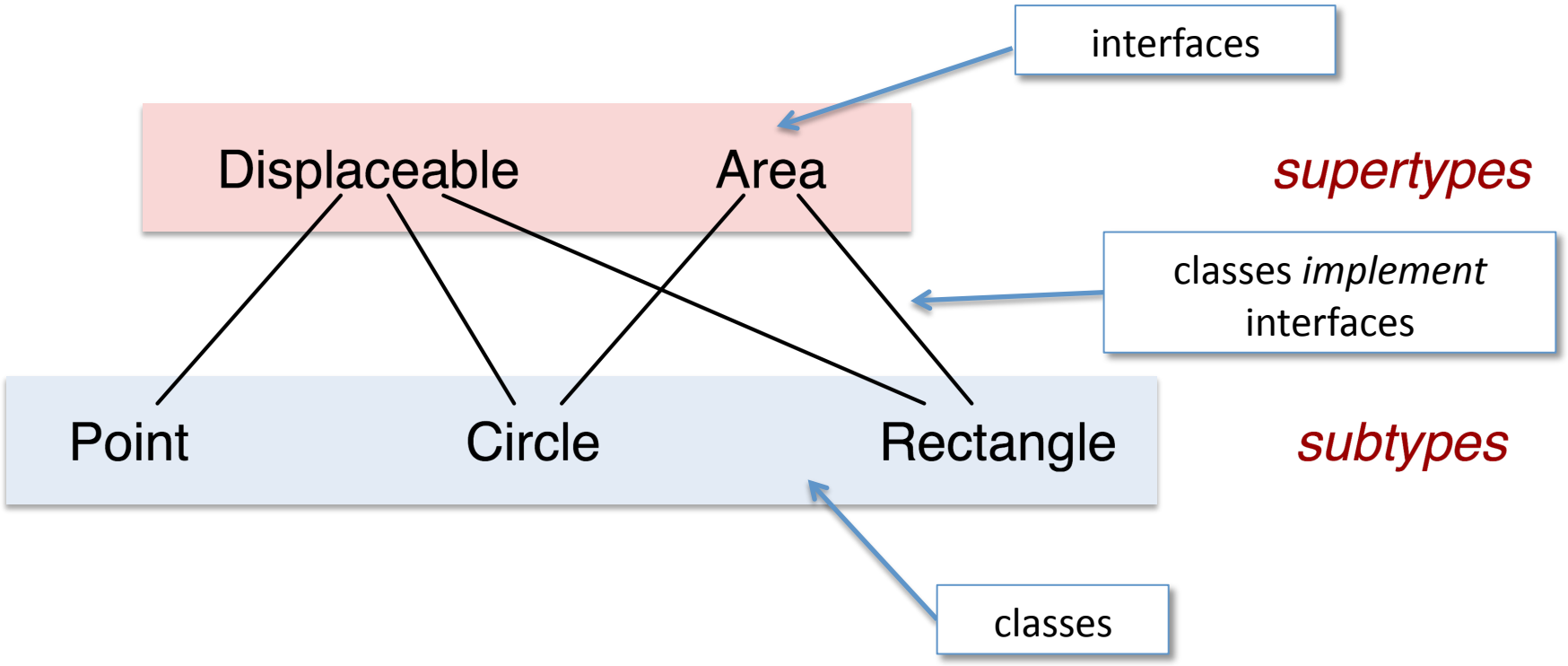
```
Area a = new Circle(1, new Point(2,3));
```

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

```
static void double m (Area x) {
   return x.getArea() * 2;
}
...
C.m( new Circle(1, new Point(2,3)) );
```

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

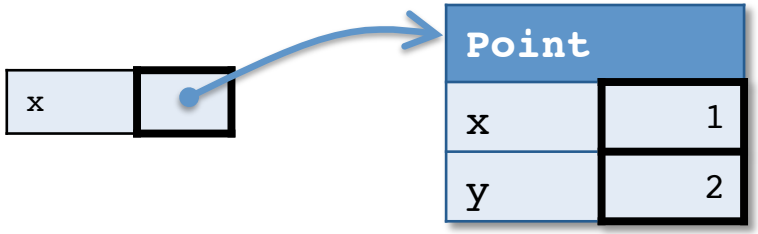
"Static" types vs. "Dynamic" classes

- The **static type** of an *expression* is a type that describes what we (and the compiler) know about the expression at compile-time (without thinking about the execution of the program)

```
Displaceable x;
```

- The **dynamic class** of an *object* is the class that it was constructed from at run time

```
x = new Point(1,2)
```



- In OCaml, we only had static types
- In Java, we also have dynamic classes because of objects
  - The dynamic class will always be a *subtype* of its static type

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

- What is the static type of s1 on line A?
- What is the dynamic class of s1 when execution reaches A?
- What is the static type of s2 on line B?
- What is the dynamic class of s2 when execution reaches B?
- What type should we declare for x (in blank D)?
- What is the dynamic class of x?
- What type should we declare for y (in blank E)?
- What is the dynamic class of y?
- Which of the assignments on lines F-I are well typed?