Programming Languages and Techniques (CIS120)

Lecture 24

March 24, 2014

Java ASM

Interfaces and Subtyping

Announcements

- HW 07 due tomorrow at midnight
- Exam 2, in class, a week from Friday (April 4th)

The Java Abstract Stack Machine

Objects and Arrays in the heap

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 *dynamic class information*
  - Arrays store *type information and length*

Heap Reference Values

Arrays

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

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

int[]			
length	4		
0	0	7	0

length *never* mutable; elements *always* mutable

Objects

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

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

Node	
elt	1
next	null

fields marked final are not mutable, all others are

What does the following program print?

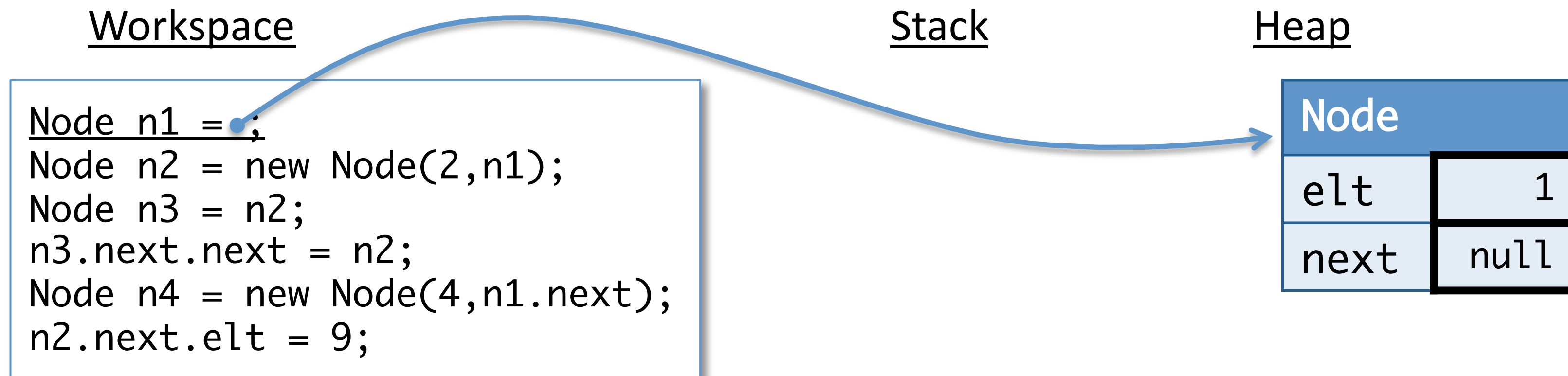
```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}

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 = 9;
    System.out.println(n1.elt);
}
```

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



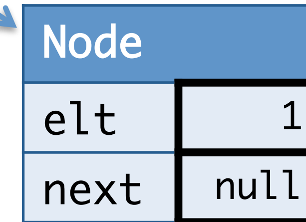
Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

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;
```

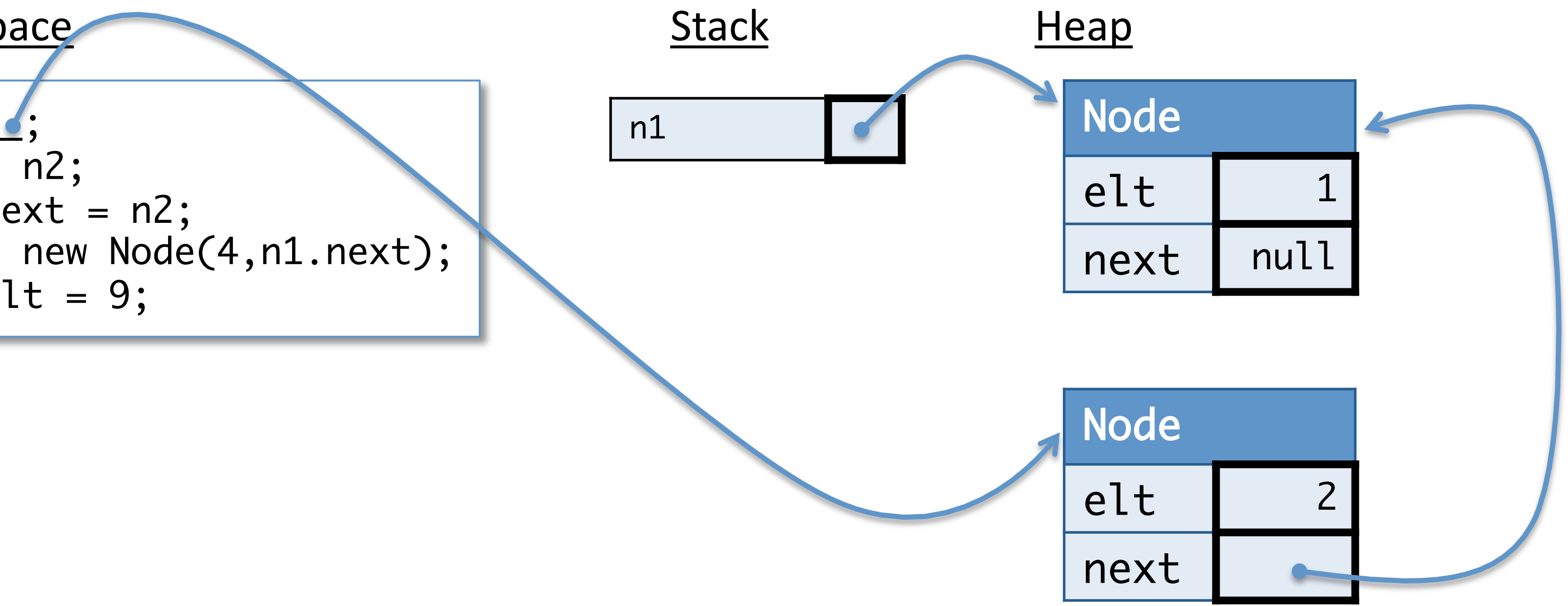
Stack

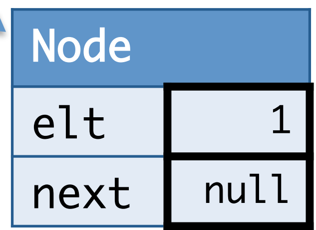
Heap

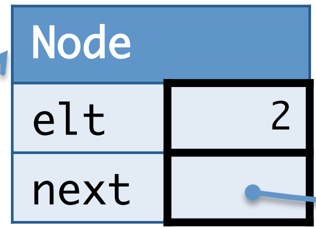


Workspace

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

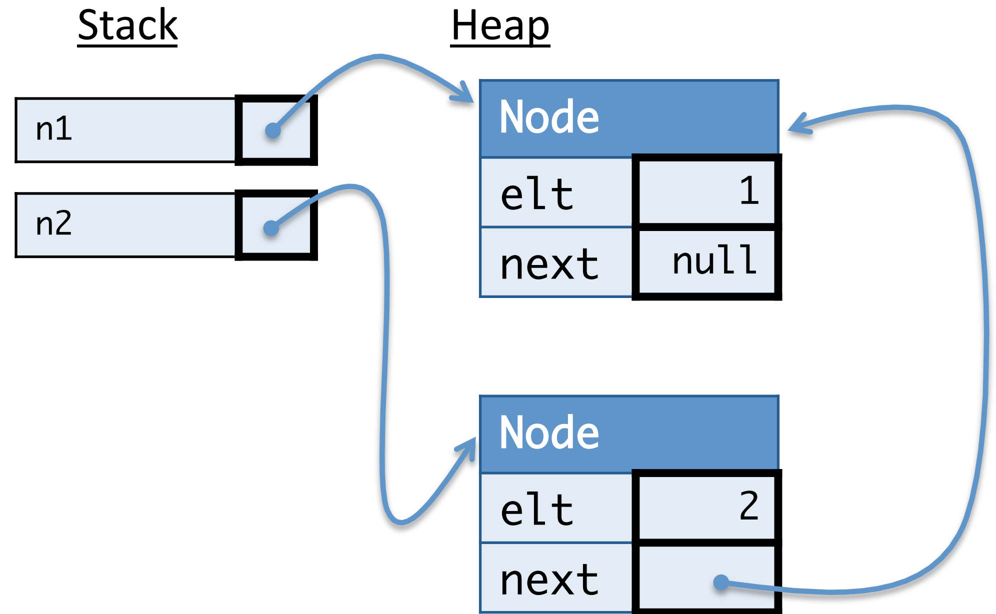






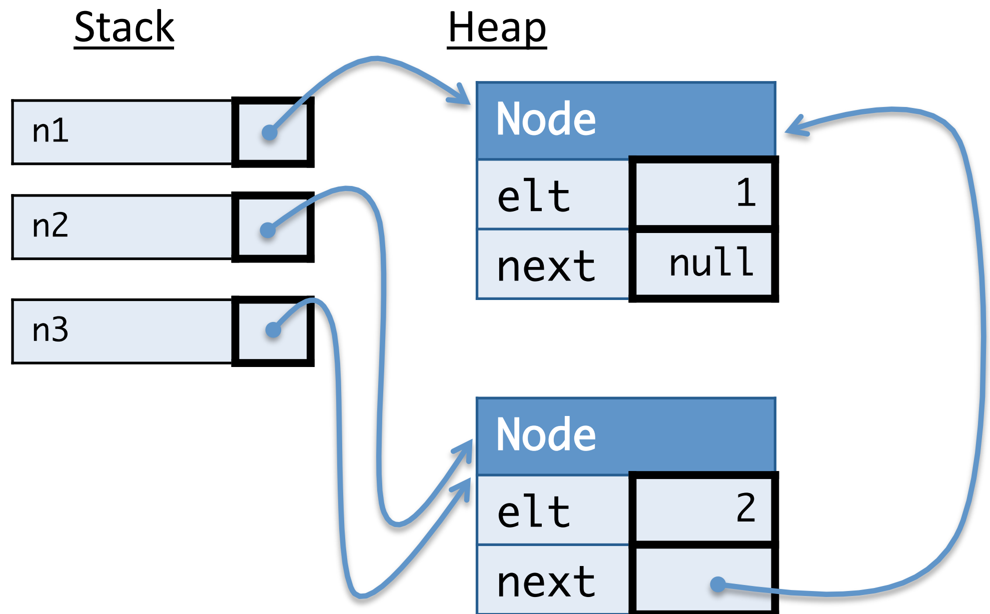
Workspace

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



Workspace

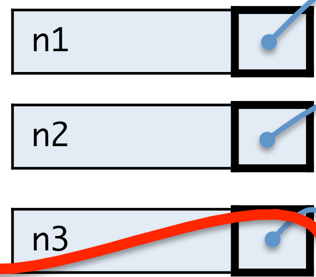
```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



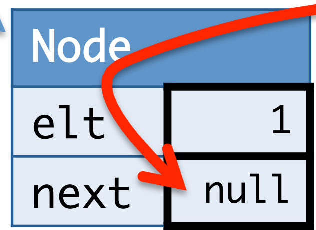
Workspace

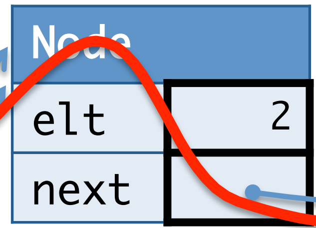
```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Stack



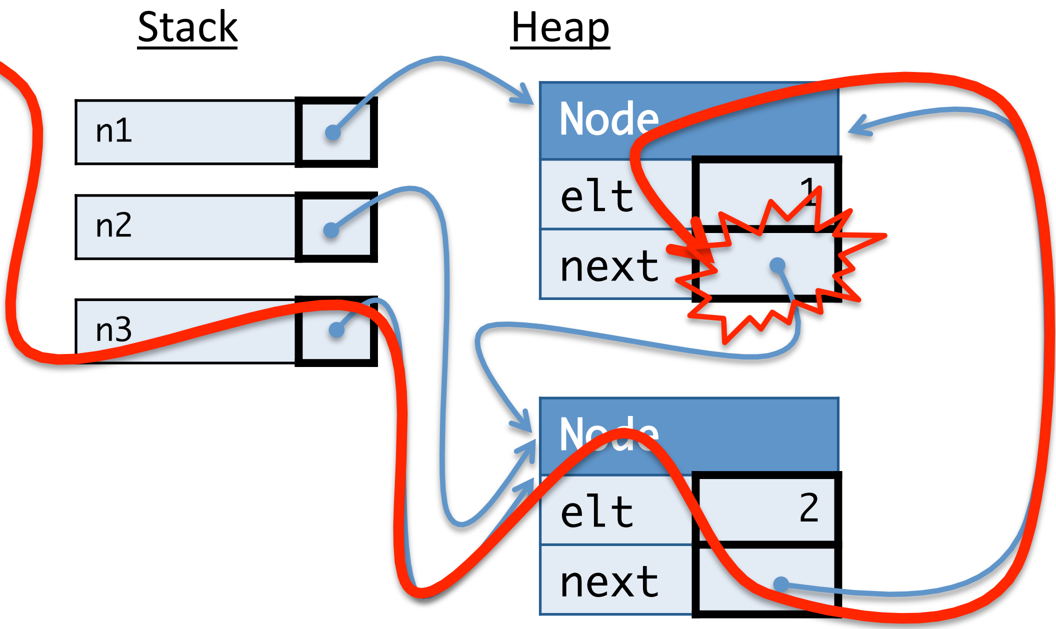
Heap

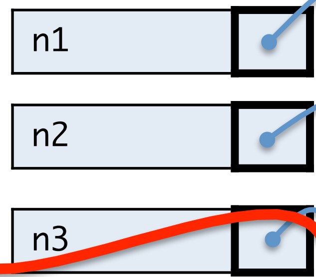


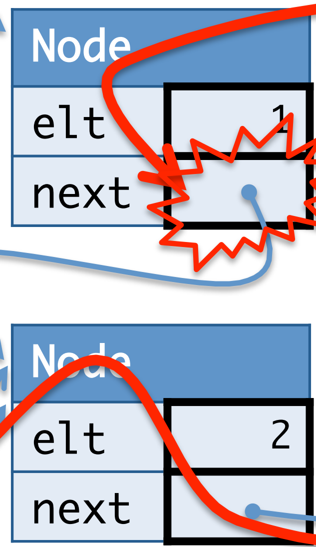


Workspace

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

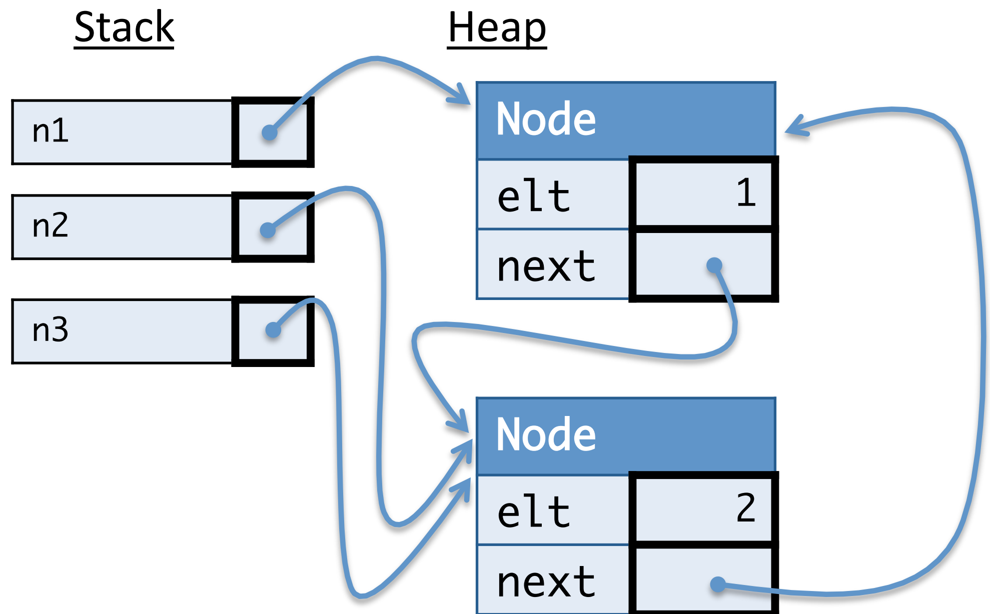






Workspace

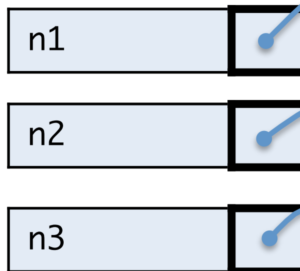
```
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



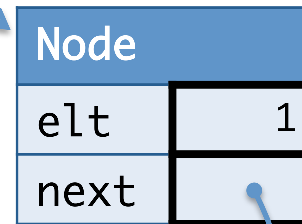
Workspace

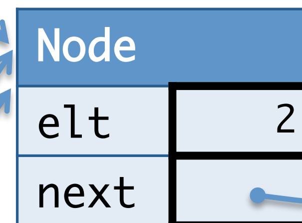
```
Node n4 = ;
n2.next.elt = 9;
```

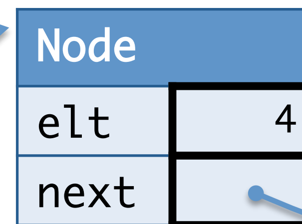
Stack



Heap

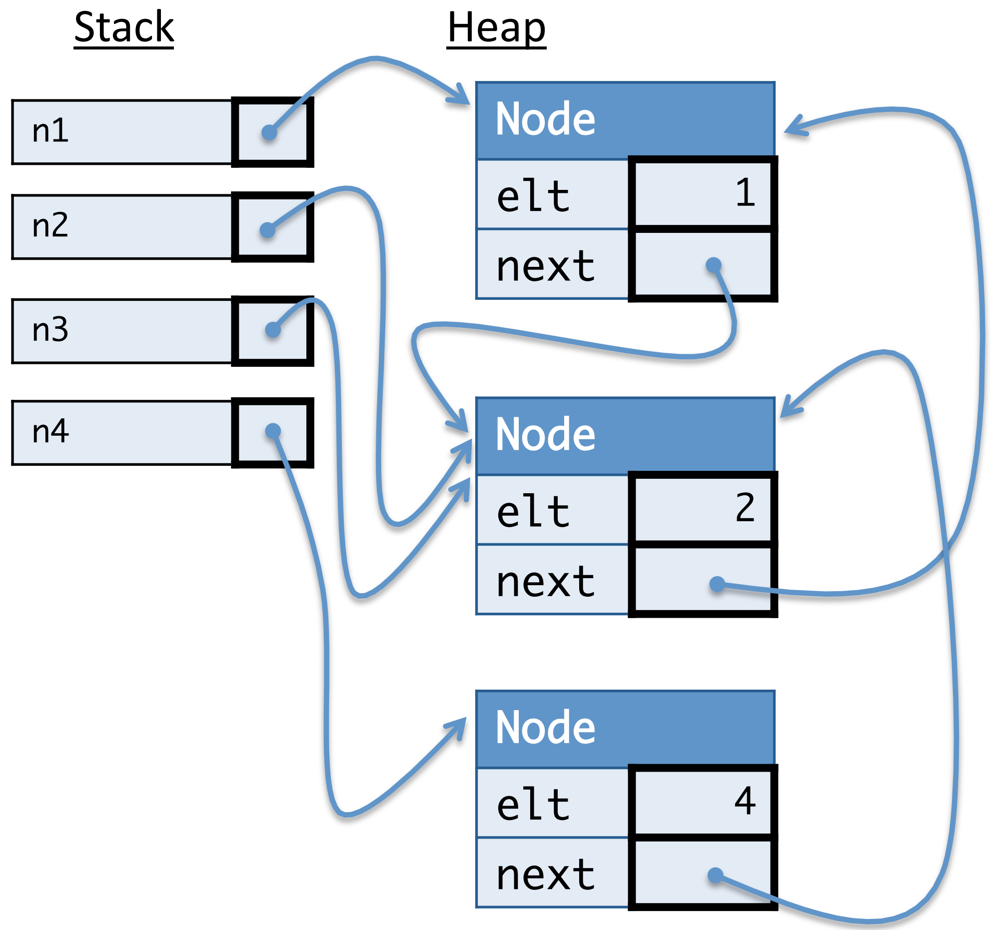


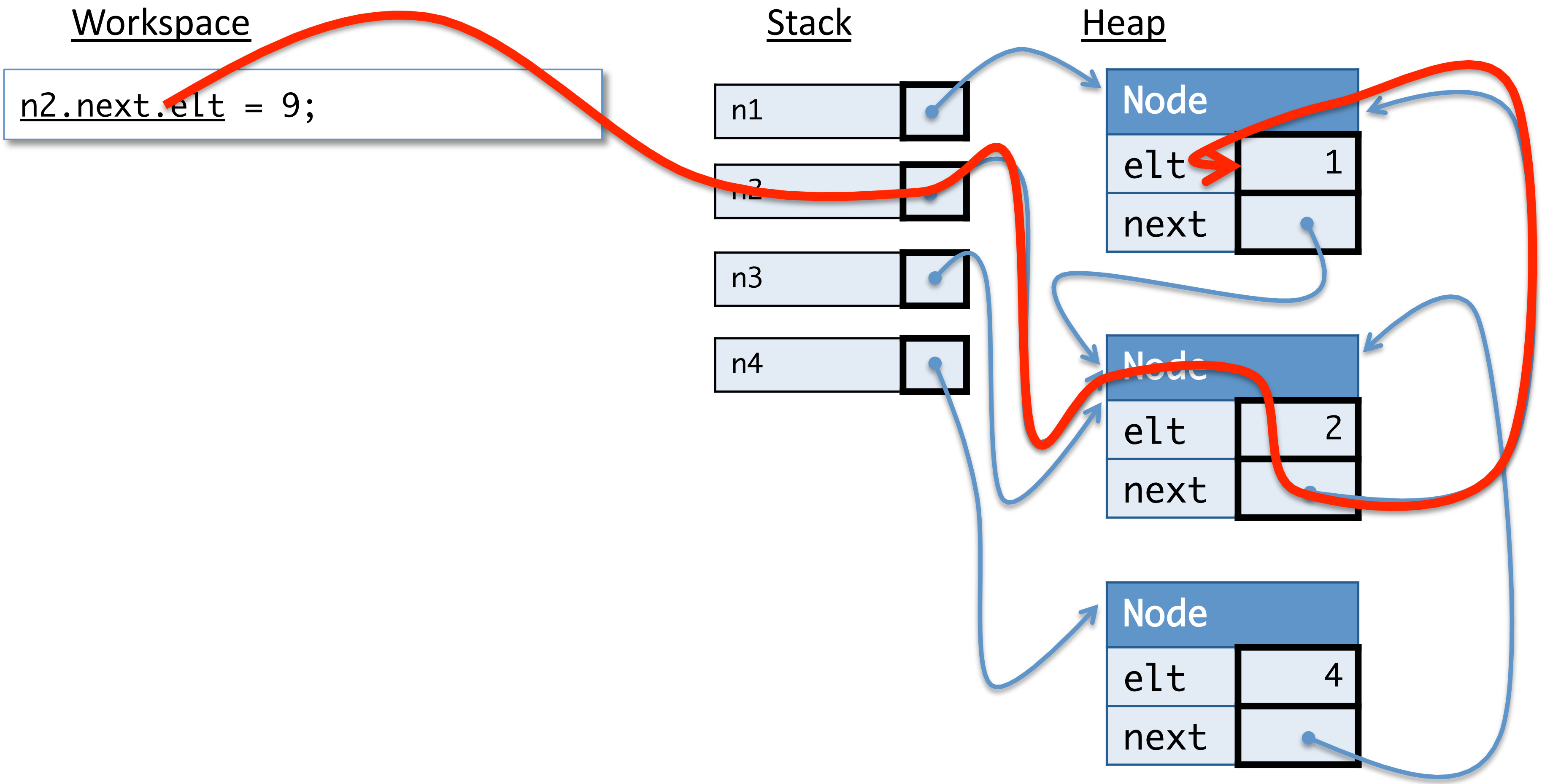


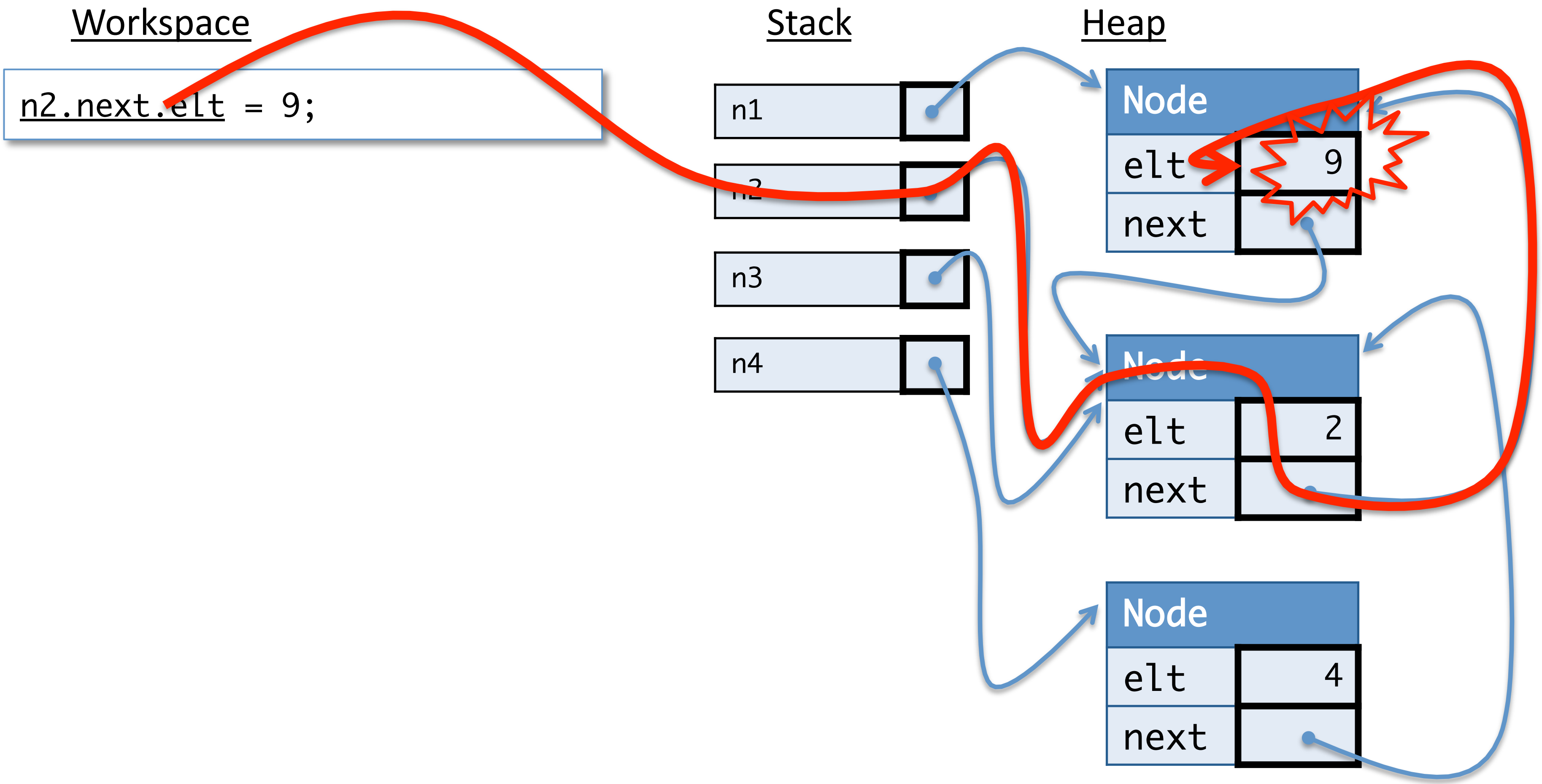


Workspace

```
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;
Stack
n1
n2
n3
n4
Heap
Node
elt
9
next
Node
elt
2
next
Node
elt
4
next

Objects and Arrays together

Preserving object invariants

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() { … }

}
```

Resizable Arrays

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

```
public class ResArray {

  private int[] data;
  private int extent;

  public ResArray() {
        data   = new int[0];
        extent = 0;
  }

  public void set(int issueNum, int numberOfCopies) {
        if (!(issueNum < data.length)) {
           // grow the array if it is too small
           // by allocating a new array for data & copying
           …
        }
        data[issueNum] = numberOfCopies;
        // update extent if it changes

  }
```

ResArray ASM

Workspace

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

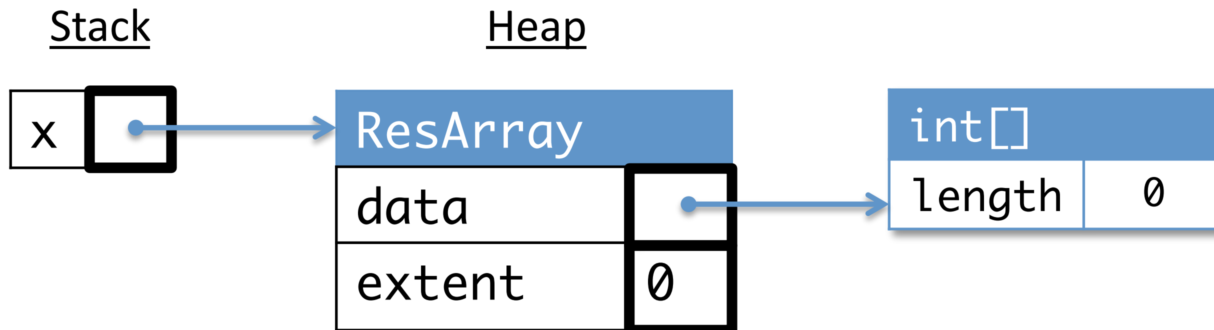
Stack

Heap

ResArray ASM

Workspace

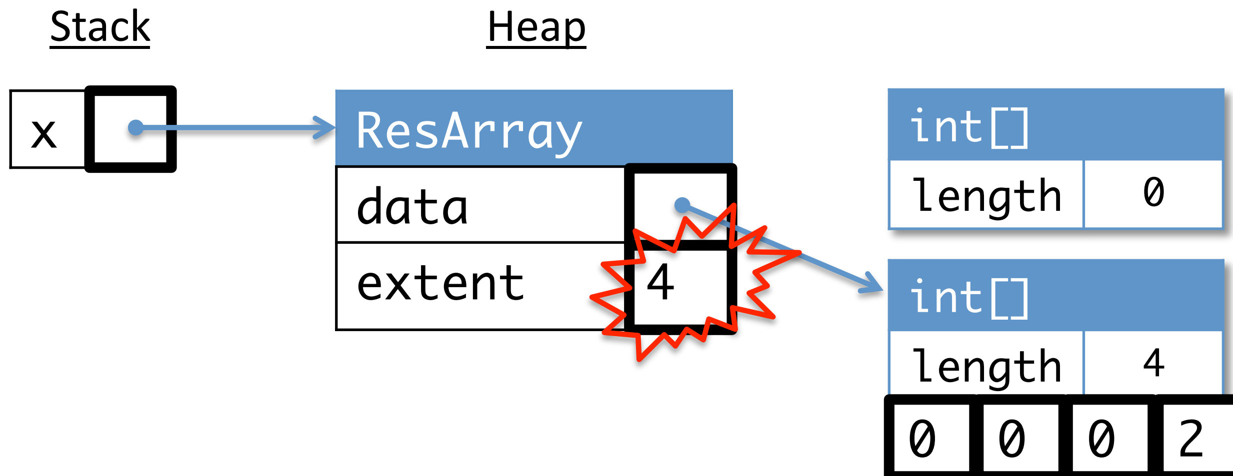
```
~~ResArray x = new ResArray();~~
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

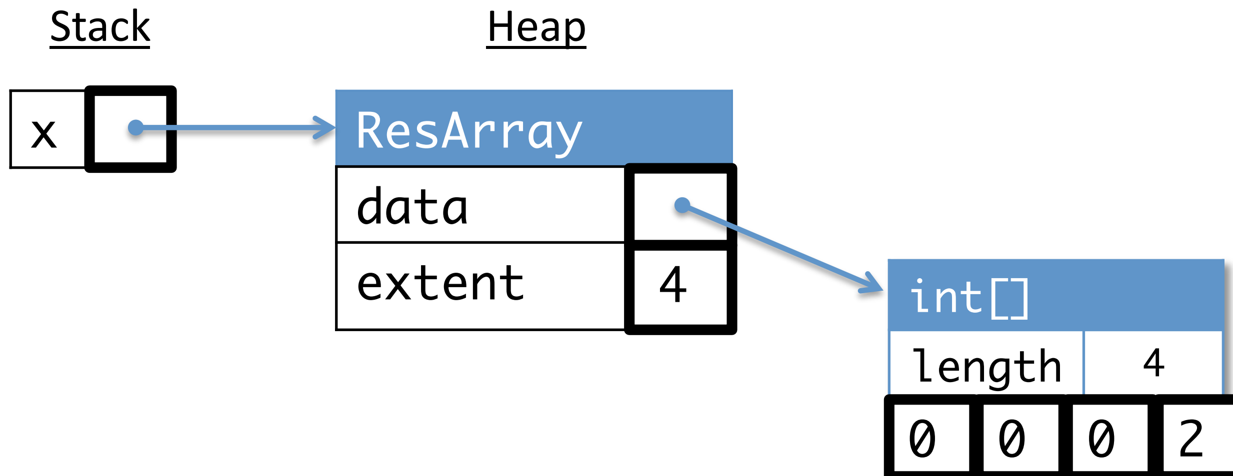
```
~~ResArray x = new ResArray();~~
~~x.set(3,2);~~
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

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



ResArray ASM

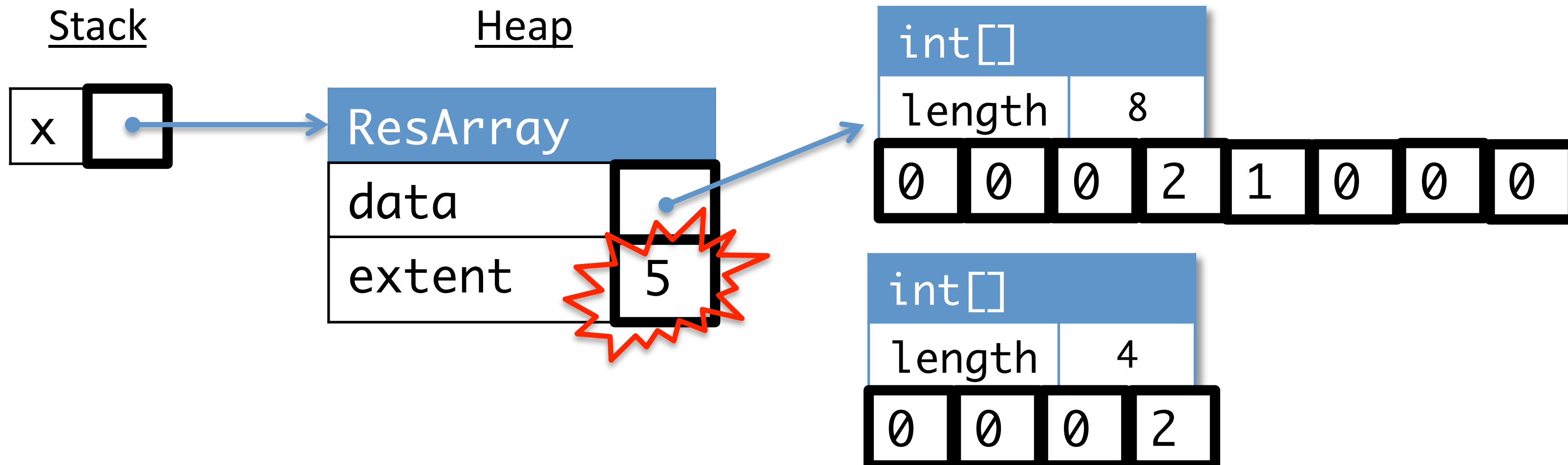
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~x.set(4,1);~~

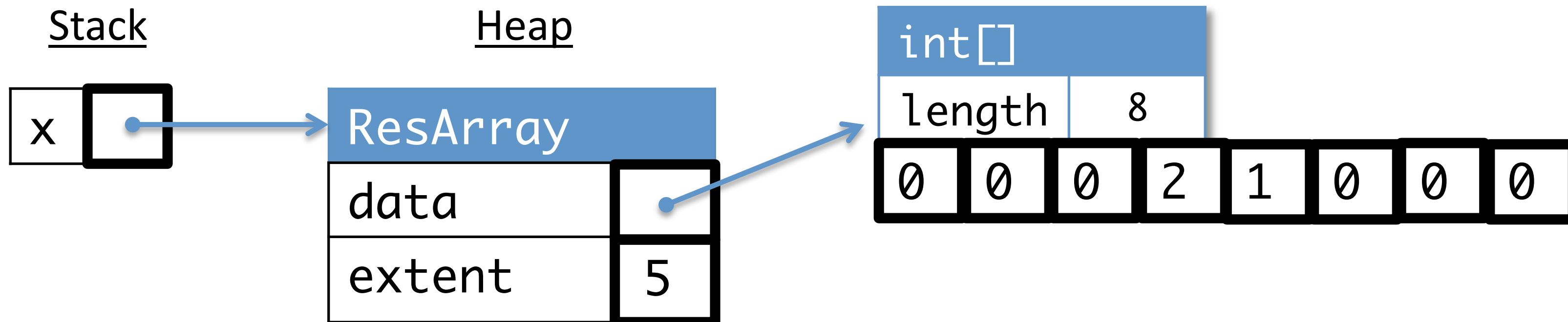
```
x.set(4,0);
```



ResArray ASM

Workspace

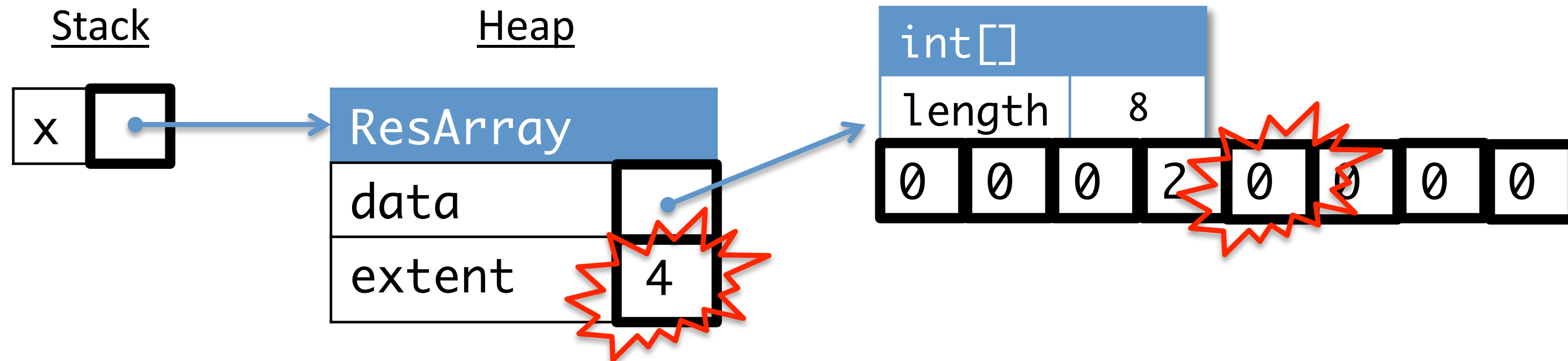
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

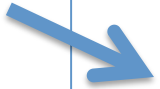
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



Resizable Arrays

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() { … }

  /** Convert the ResArray to a normal array.
   */
  public int[] values() { … }
}
```



Values Method

Which implementation is better?

1.
```
public int[] values() {
    int[] values = new int[extent];
    for (int i=0; i < extent; i++) {
        values[i] = data[i];
    }
    return values;
}
```

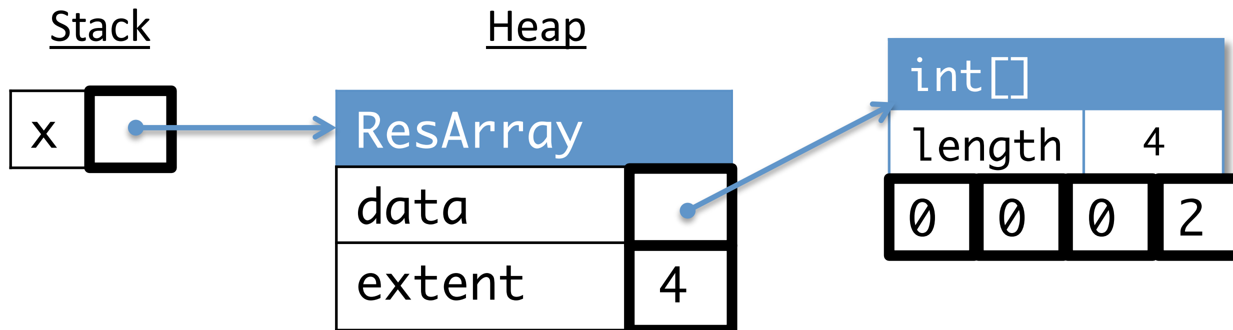
2.
```
public int[] values() {
    return data;
}
```

3. Neither one. They are both about the same.

ResArray ASM

Workspace

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



ResArray ASM

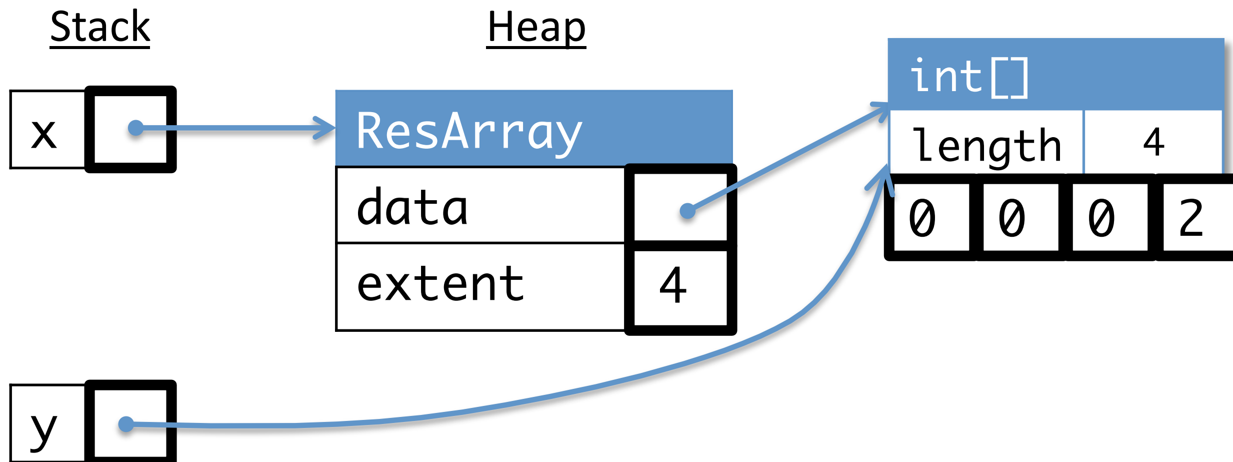
Workspace

~~ResArray x = new ResArray();~~

~~x.set(3,2);~~

~~int[] y = x.values();~~

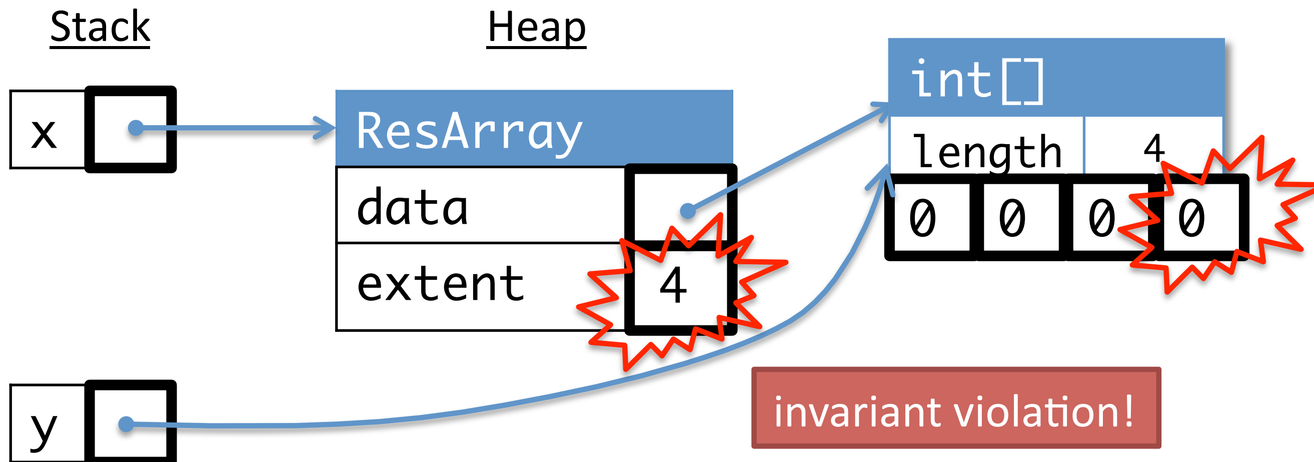
y[3] = 0;



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.

Interfaces and Subtyping

Working with objects abstractly

Java “Objects” in OCaml

```
(* The type of counter
   “objects” *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

(* Create a counter "object"
with hidden state: *)
let new_counter () : counter =
  let r = {contents = 0} in {
    inc = (fun () ->
       r.contents <-
          r.contents + 1;
       r.contents);
    dec = (fun () ->
       r.contents <-
          r.contents - 1;
       r.contents)
}
```

Type is separate from the implementation

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

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

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Class specifies both type and implementation of object values

Java “Objects” in OCaml

```
(* Type of GUI elements *)

type widget = {
    repaint : gctx -> unit;
    handle  : gctx -> int;
    size    : gctx -> dimension
}


let label (s: string) : widget
* label_controller =
  let r = {contents = s} in
  {
    repaint = …;
    handle  = …;
    size    = …;
  },
  {
   set_label = …
  }
```

Type unifies different sorts of GUI elements

```
public class Label {

  private String r;

  public Label(String r0) {
    r = r0;
  }

  public void setLabel(String s) {
    r = s;
  }

  public void repaint (Gctx g) {
    …
  }

  public void handle (Gctx g) {
    …
  }
}
```

What about widget?

Interfaces

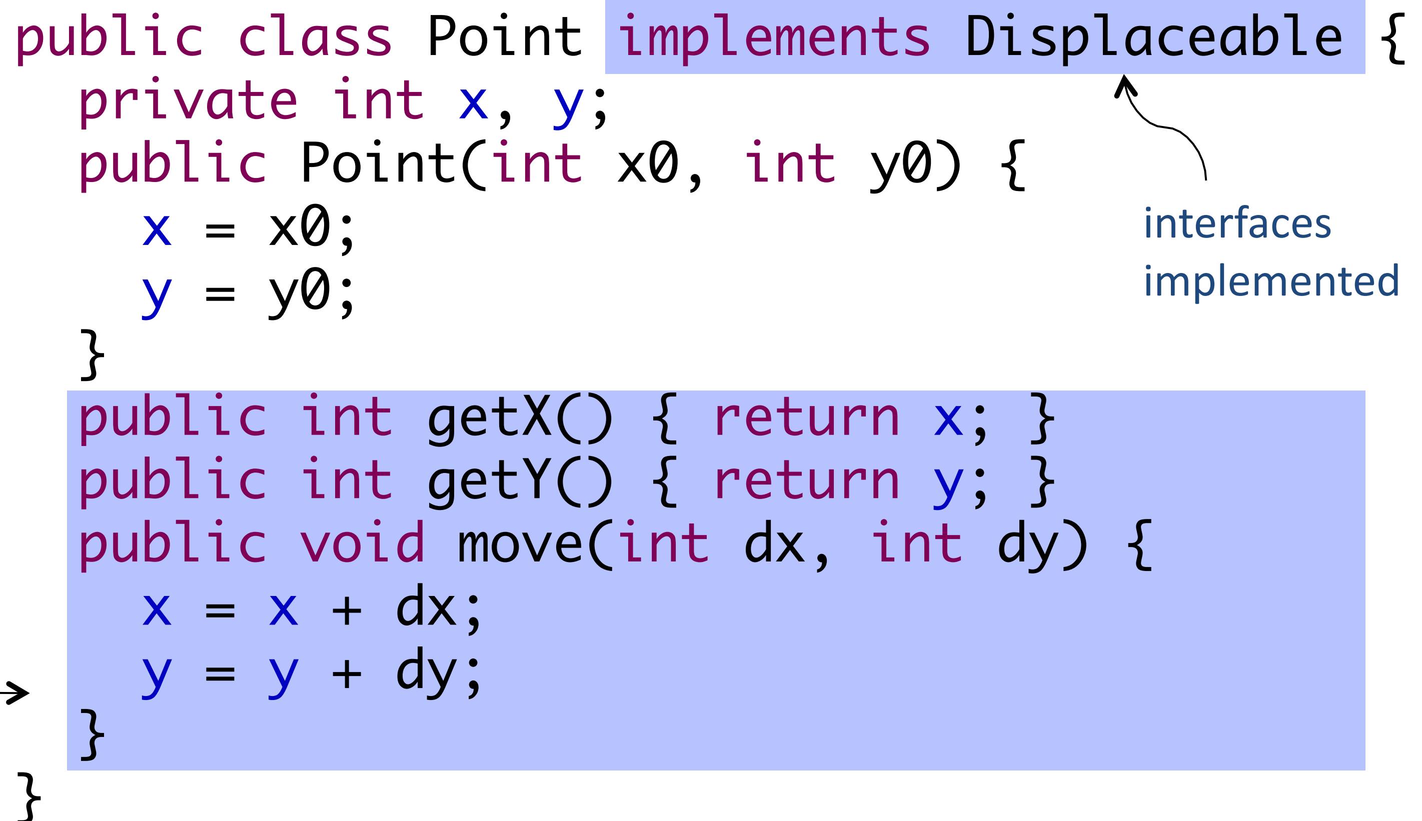
- Give a type for an object based on what it *does*, not on how it was constructed
- A "signature" for objects, describing a collection of methods that must be provided by classes that *implement* the interface
- Example:

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

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface
- That class fulfills the contract implicit in the interface



Another implementation

```
public class Circle implements Displaceable {
  private Point center;
  private int radius;
  public Circle(Point initCenter, int initRadius) {
    center = initCenter;
    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

Delegation: move the circle by moving the center

And another…

```
class  ColoredPoint implements Displaceable {
  private Point p;
  private Color c;
   ColoredPoint (int x0, int y0, Color c0) {
     p = new Point(x0,y0);
     c = c0;
}
  public void move(int dx, int dy) {
     p.move(dx, dy);
  }
  public int getX() { return p.getX(); }

  public int getY() { return p.getY(); }

  public Color getColor() { return c; }
}
```

Flexibility: Classes may contain more methods than interface requires

Multiple interfaces

- An interface represents a point of view

  ...but there can be multiple valid points of view

- Example:
  - All shapes can move (all are Displaceable)
  - Some are two dimensional (have Area)

Area interface

- Contract for objects that are 2 Dimensional
  - Circles have Area
  - Points don’t

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

Circles with Area

```
public class Circle implements Displaceable, Area {
  private Point center;
  private int radius;
  public Circle(Point initCenter, int initRadius) {
    center = initCenter;
    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);
  }
  public double getArea ()  {
       return 3.14159 * r * r;
  }
}
```

Types and Subtyping

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 int r;
    private Point p;
    public Circle (Point p0, int r0) {
        r = r0; p = p0;
    }
    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));
```

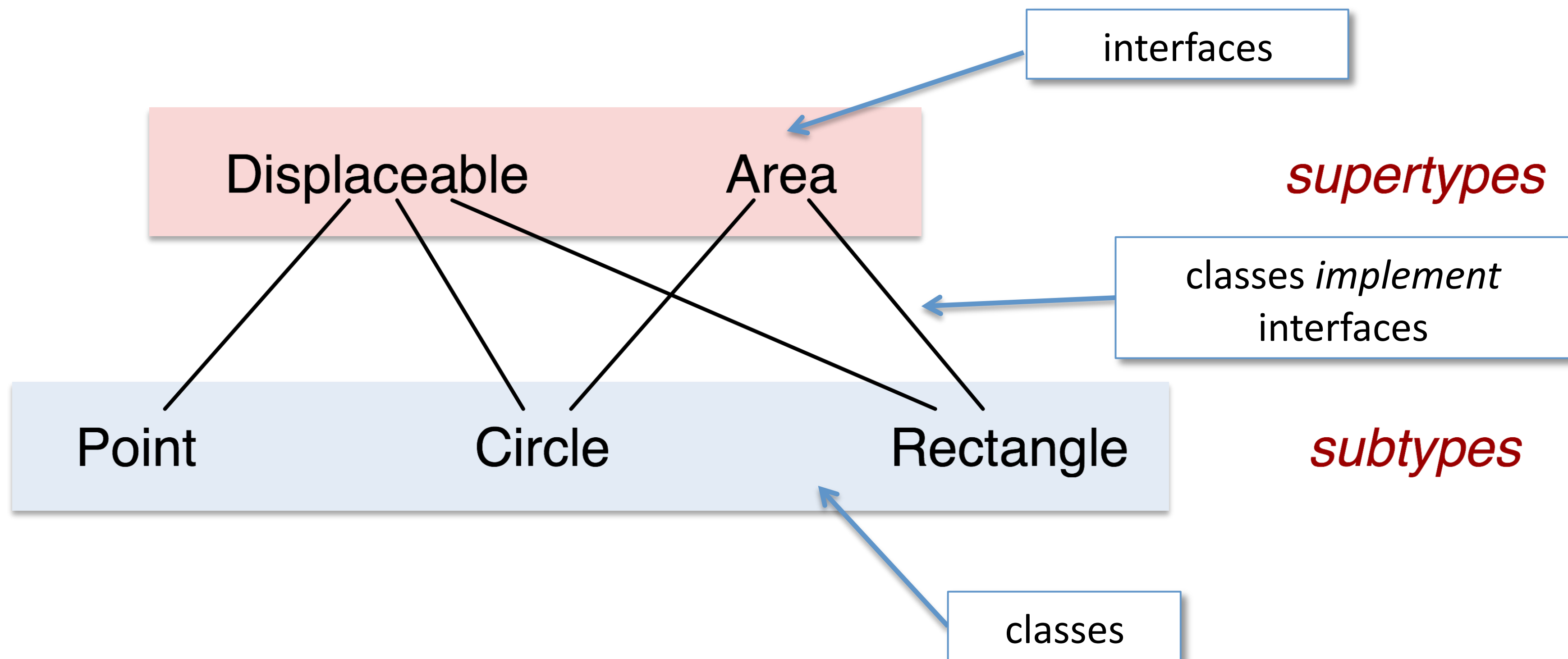
supertype of Circle — subtype of Area

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

```
static 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

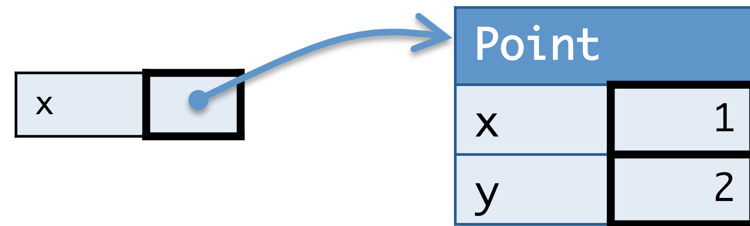
"Static" types vs. "Dynamic" classes

- 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
  - The dynamic class will always be a *subtype* of its static type