Programming Languages and Techniques (CIS120)

Lecture 25

Java ASM, Subtyping

Chapter 32 & Chapter 22

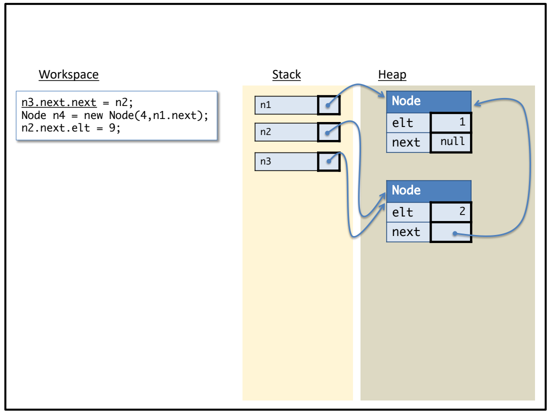
Revenge of the Son of the Abstract Stack Machine

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 values: objects and arrays



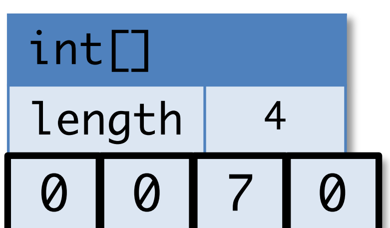
- Key differences:
  - Everything, including stack slots, is mutable by default
  - Objects store dynamic class information: *what class was used to create them*
  - *Arrays store type information and length field*
  - *New component: Class table (coming soon)*

Heap Reference Values

Arrays

- Type of values that it stores
- Length field (immutable)
- Values for all of the array elements

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

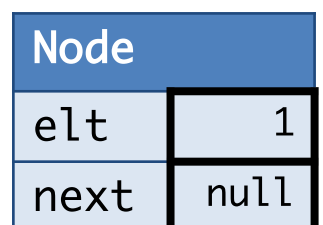


length *never* mutable; elements *always* mutable

Objects

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

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



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

ResArray ASM

Workspace

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

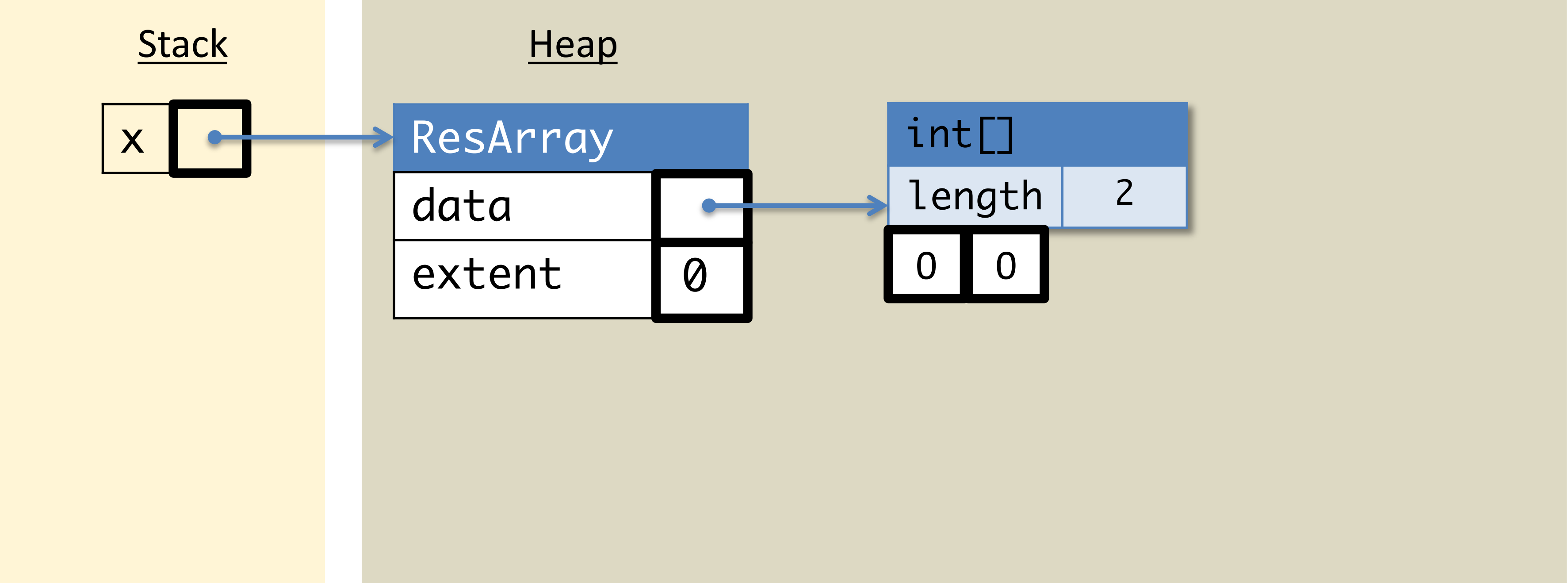
Stack

Heap

ResArray ASM

Workspace

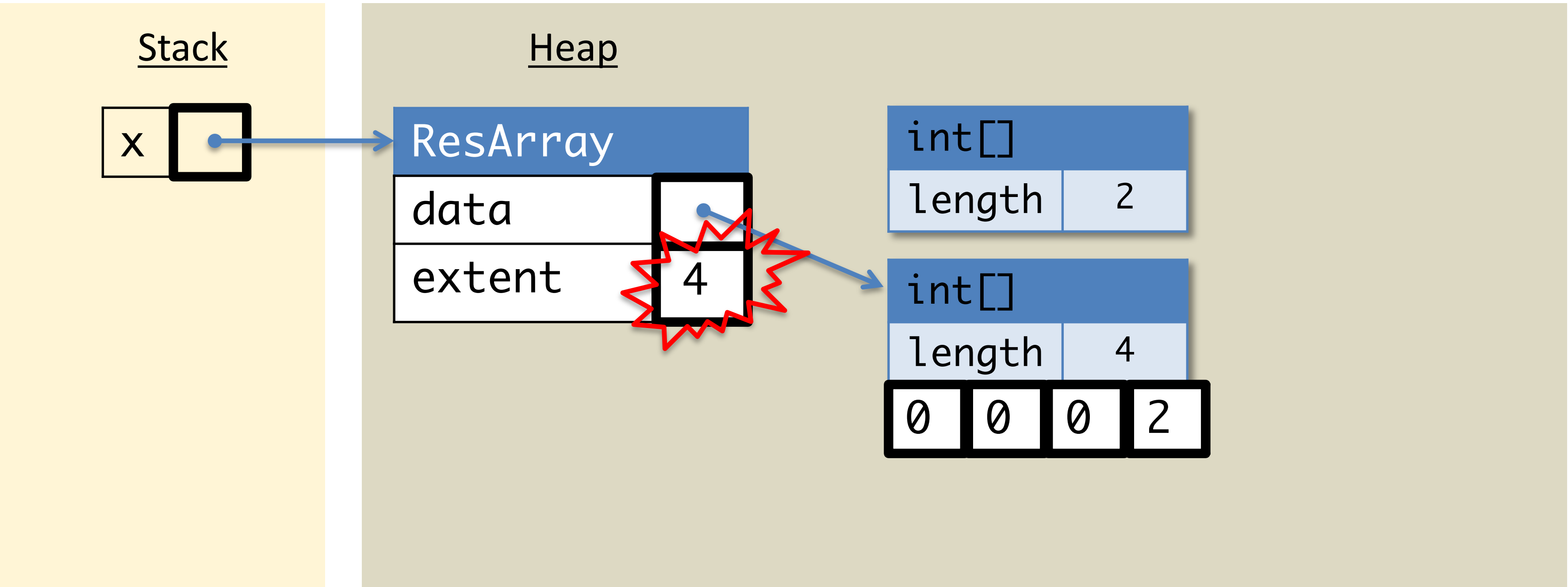
```
~~ResArray x = new ResArray();~~
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

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



ResArray ASM

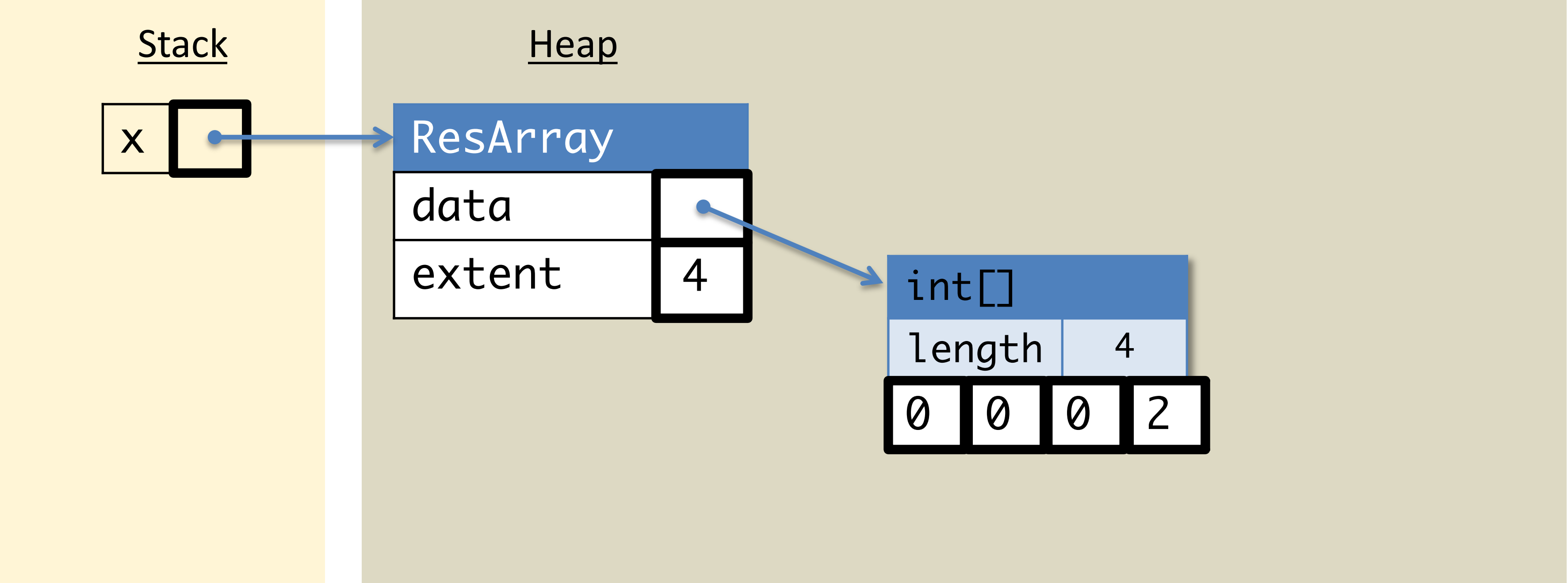
Workspace

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

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

x.set(4,1);

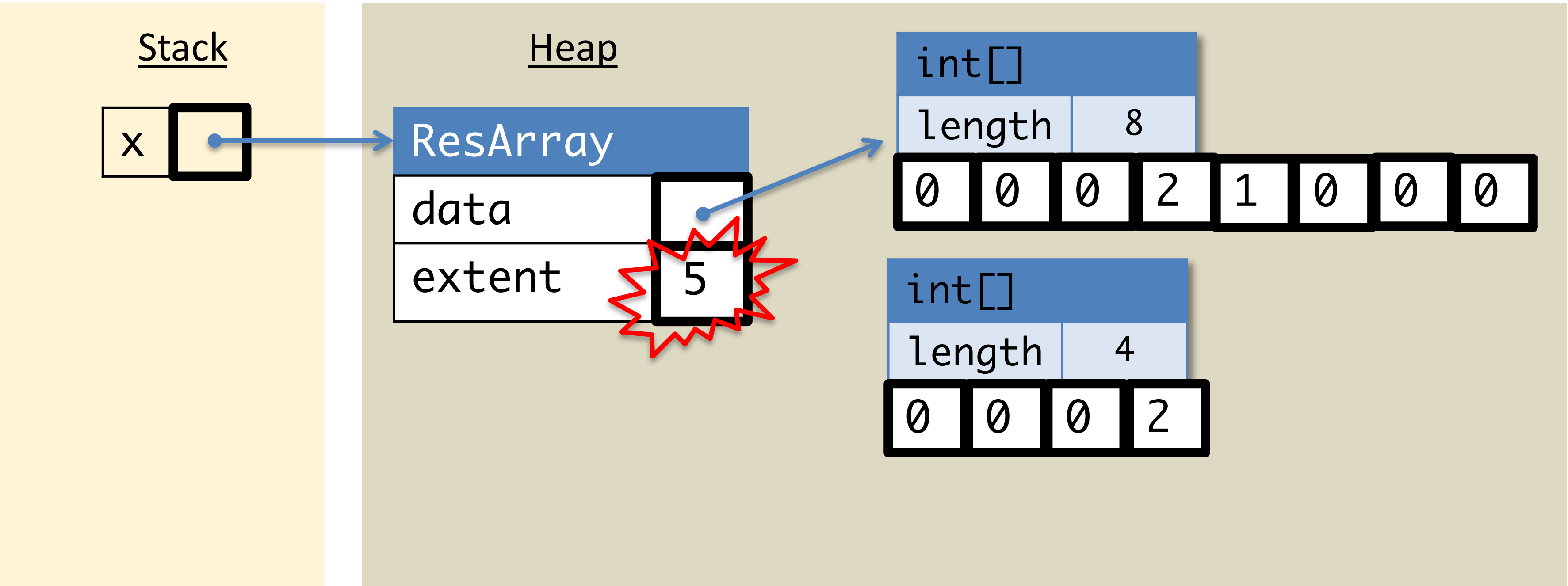
x.set(4,0);



ResArray ASM

Workspace

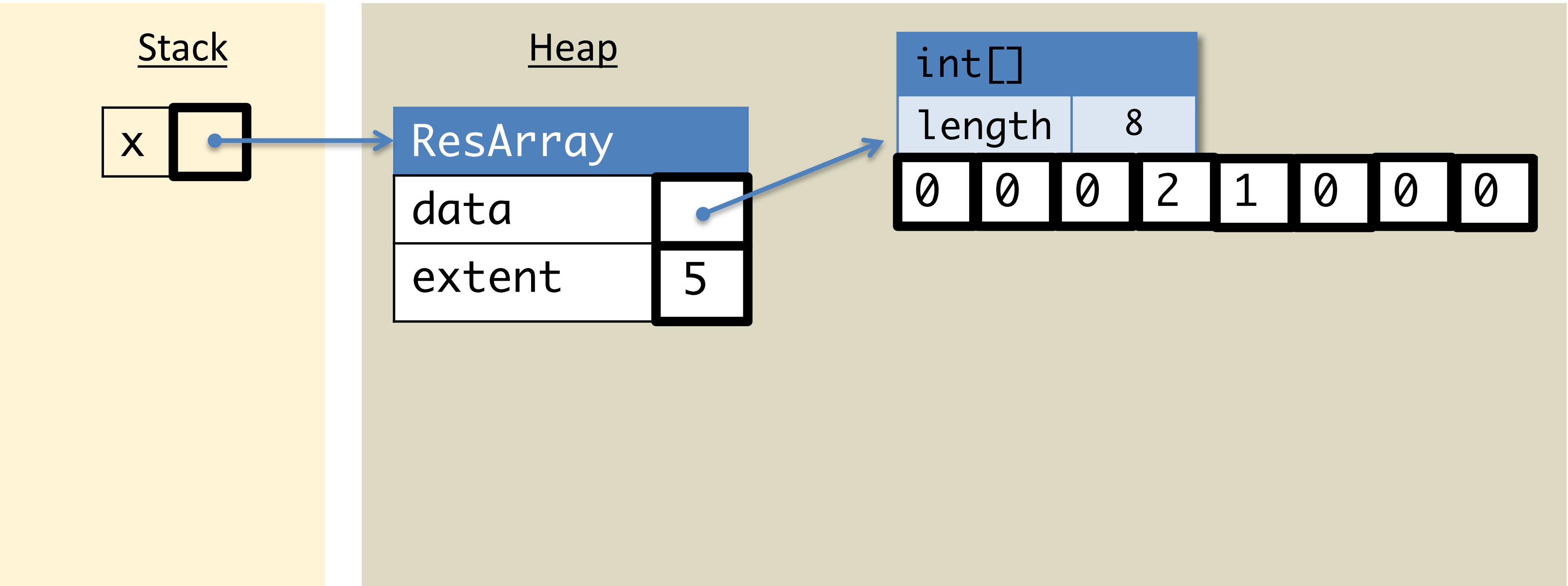
```
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
```



ResArray ASM

Workspace

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



ResArray ASM

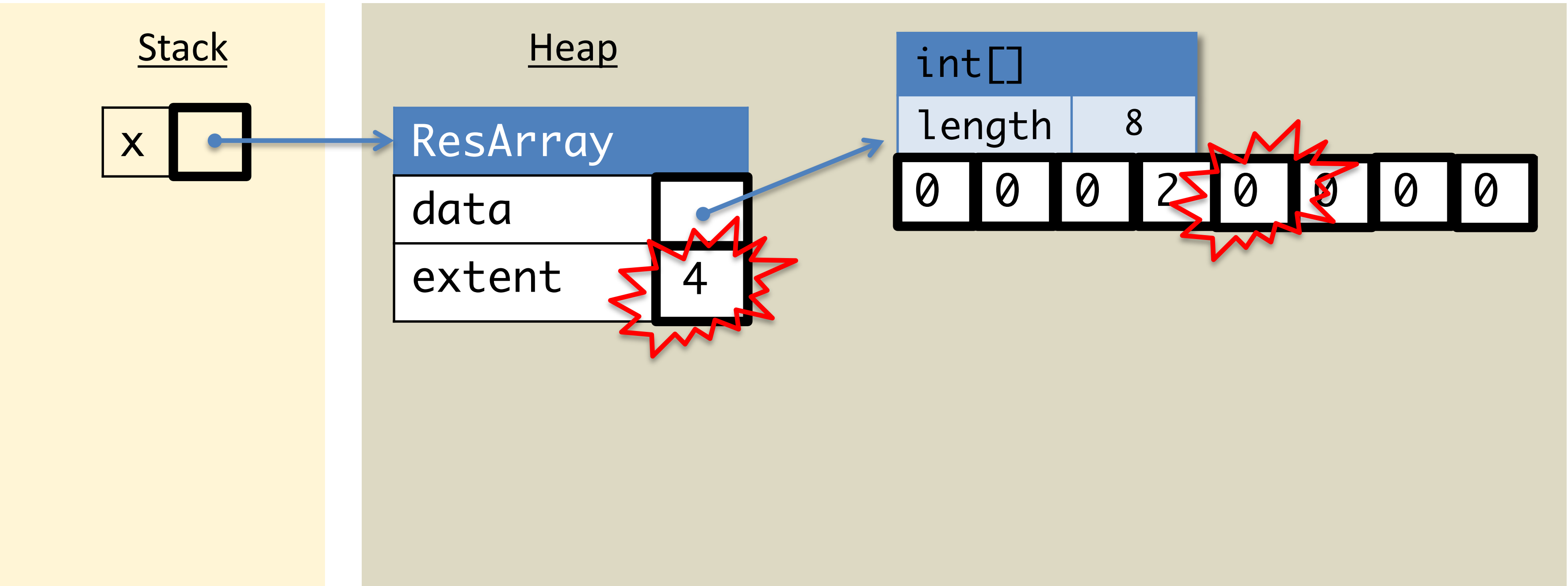
Workspace

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

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

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

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



Refactor 3: Optimize values

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

Or maybe we can do it this way? …

```
public int[] values() {
    if (data.length == extent) {
      return data;
    } else {
      // as above
    }
}
```

ResArray ASM

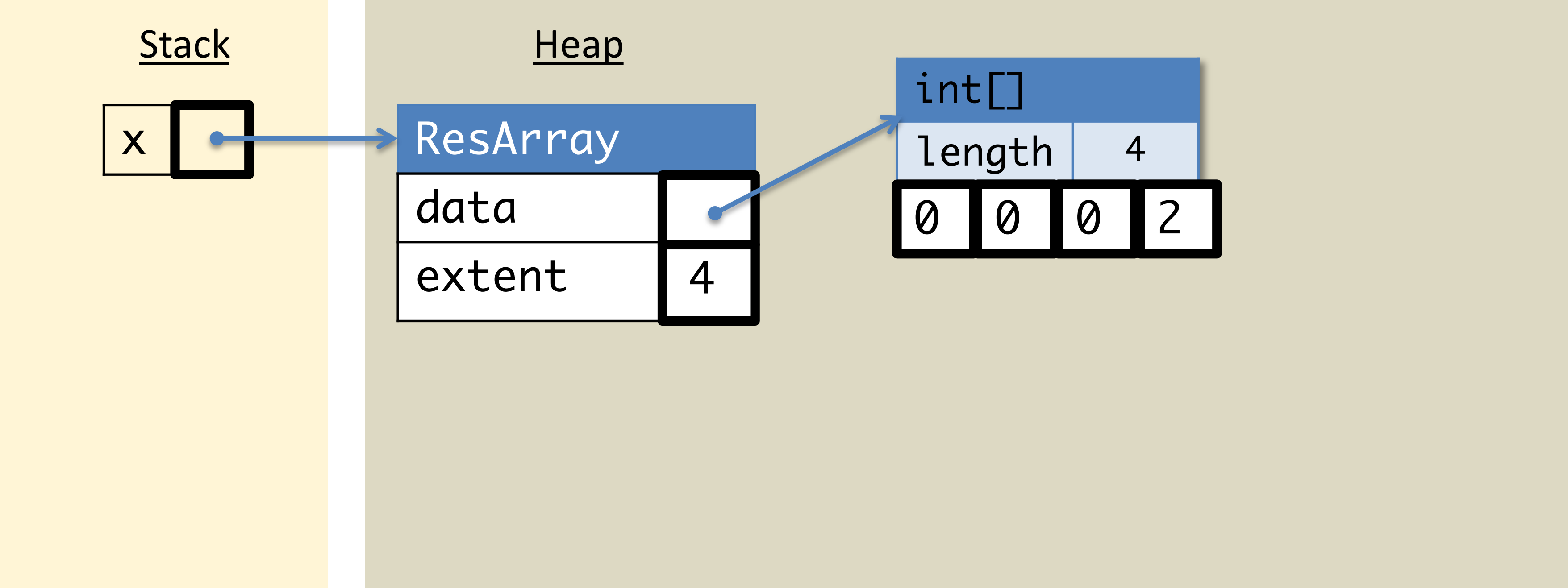
Workspace

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

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

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

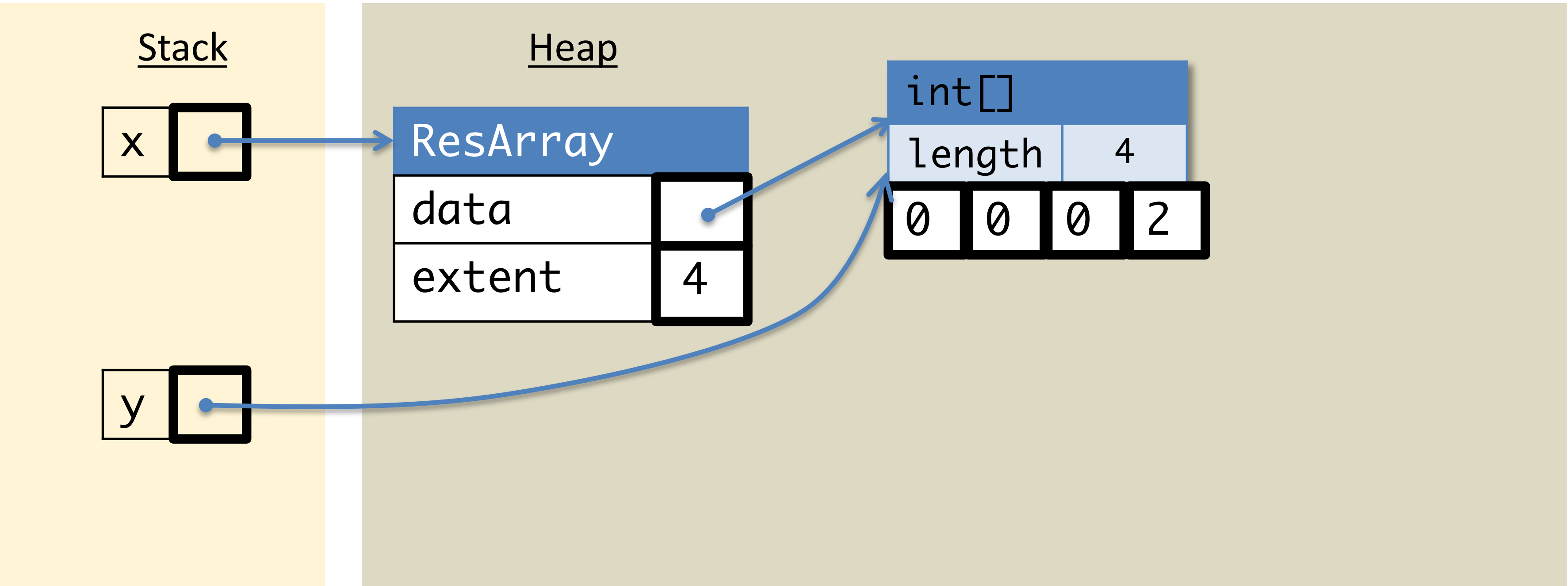
`y[3] = 0;`



ResArray ASM

Workspace

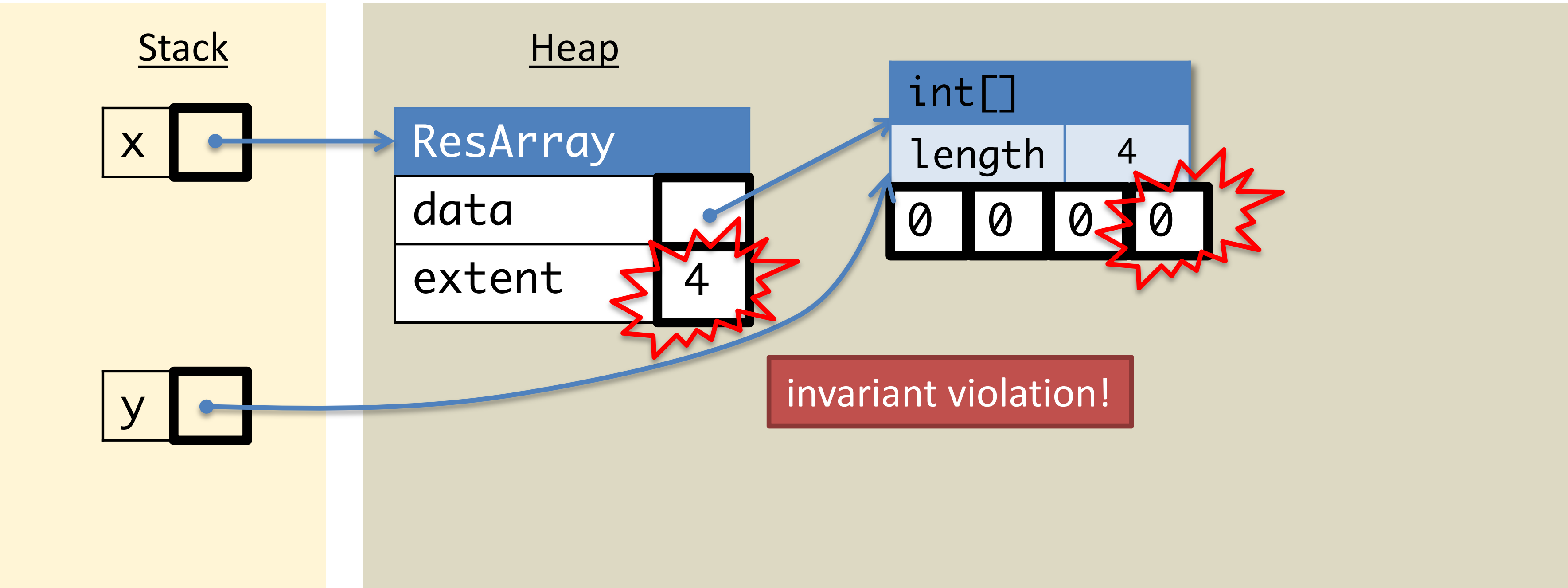
```
ResArray x = new ResArray();
x.set(3,2);
int[] y = x.values();
y[3] = 0;
```



ResArray ASM

Workspace

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



Objects in the ASM

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

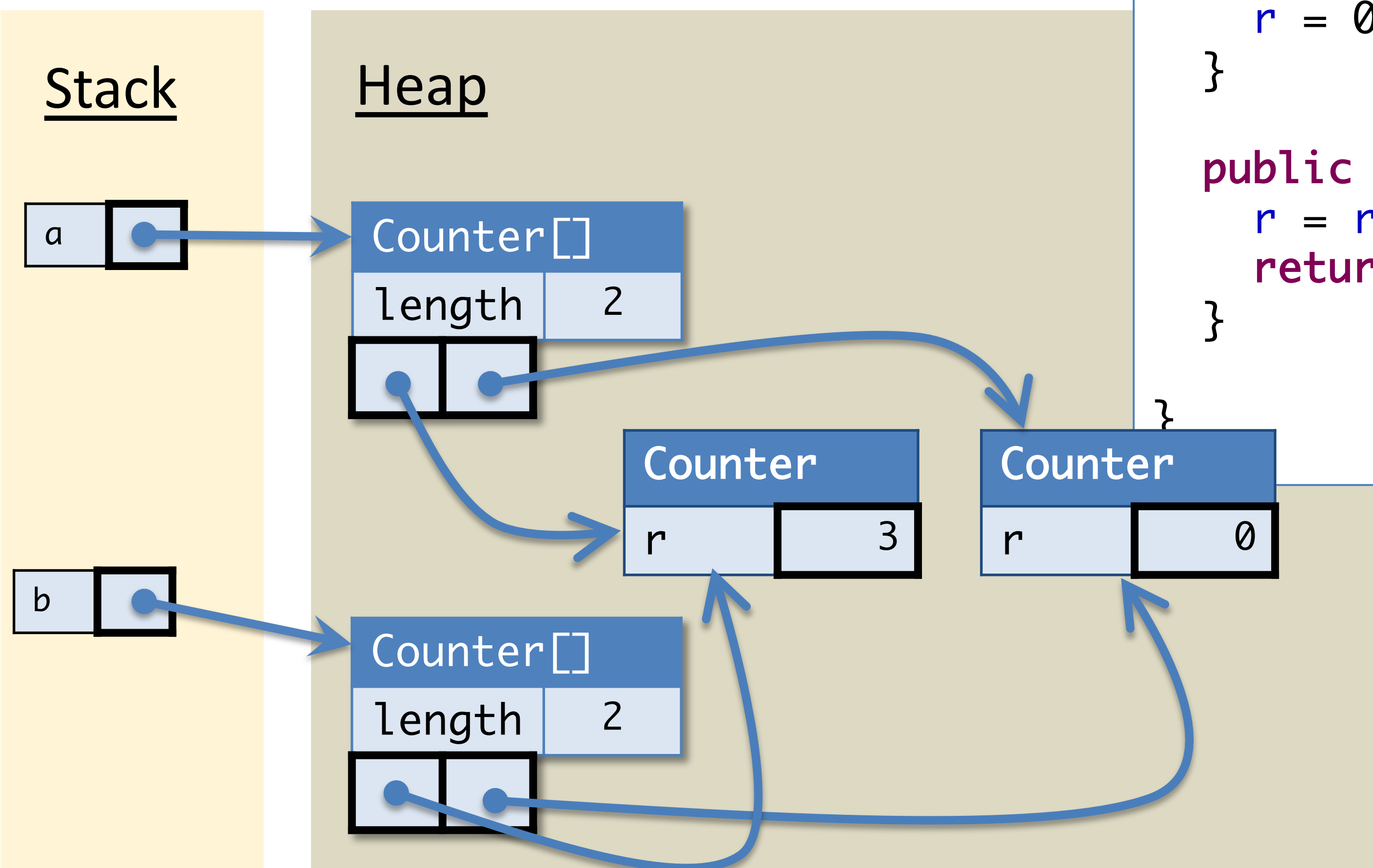
```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```
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?

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

```java
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

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

}
```



What does the following program print?
1 – 9

or 0 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

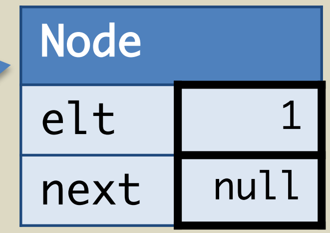
Heap

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

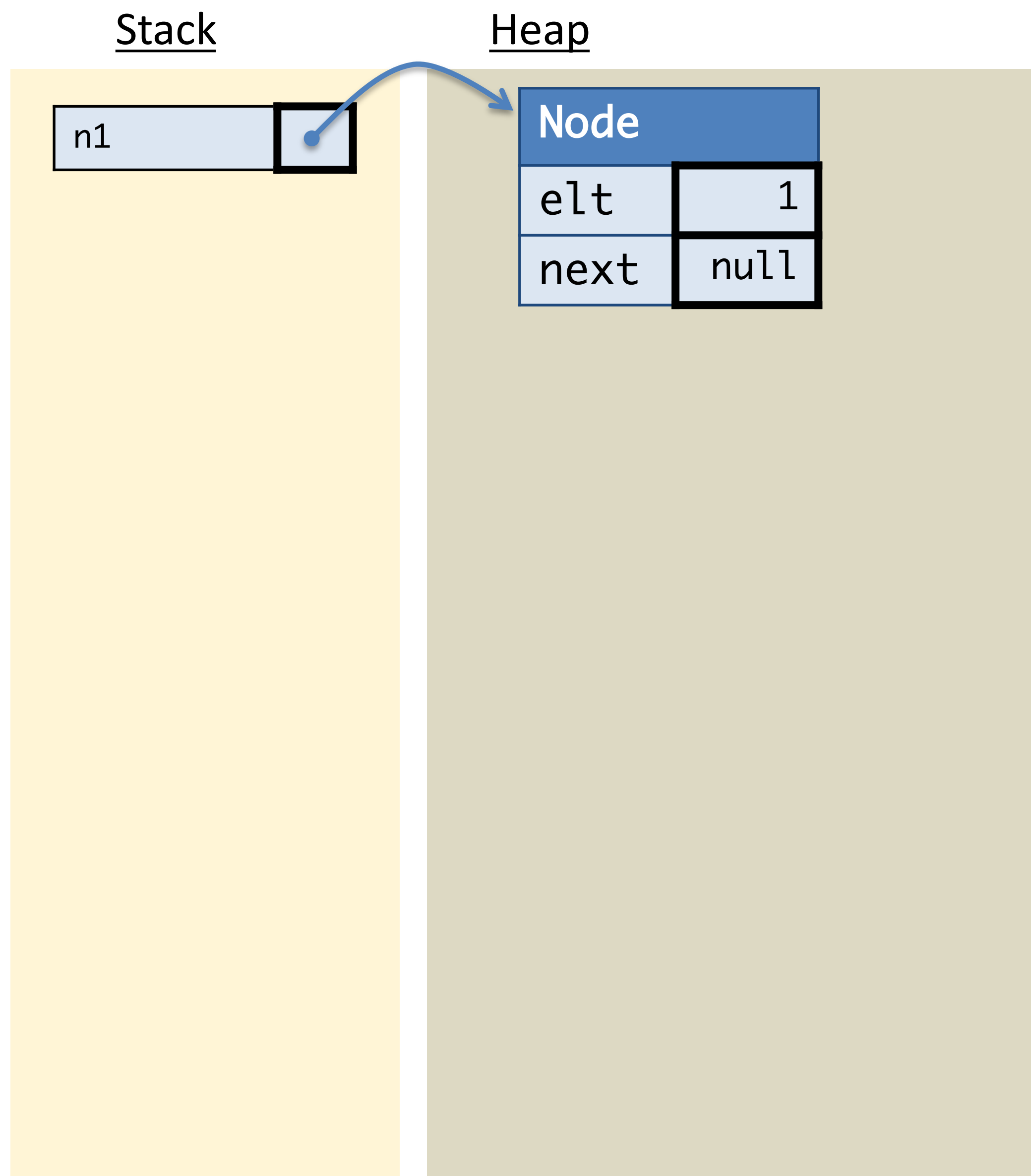
Heap



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

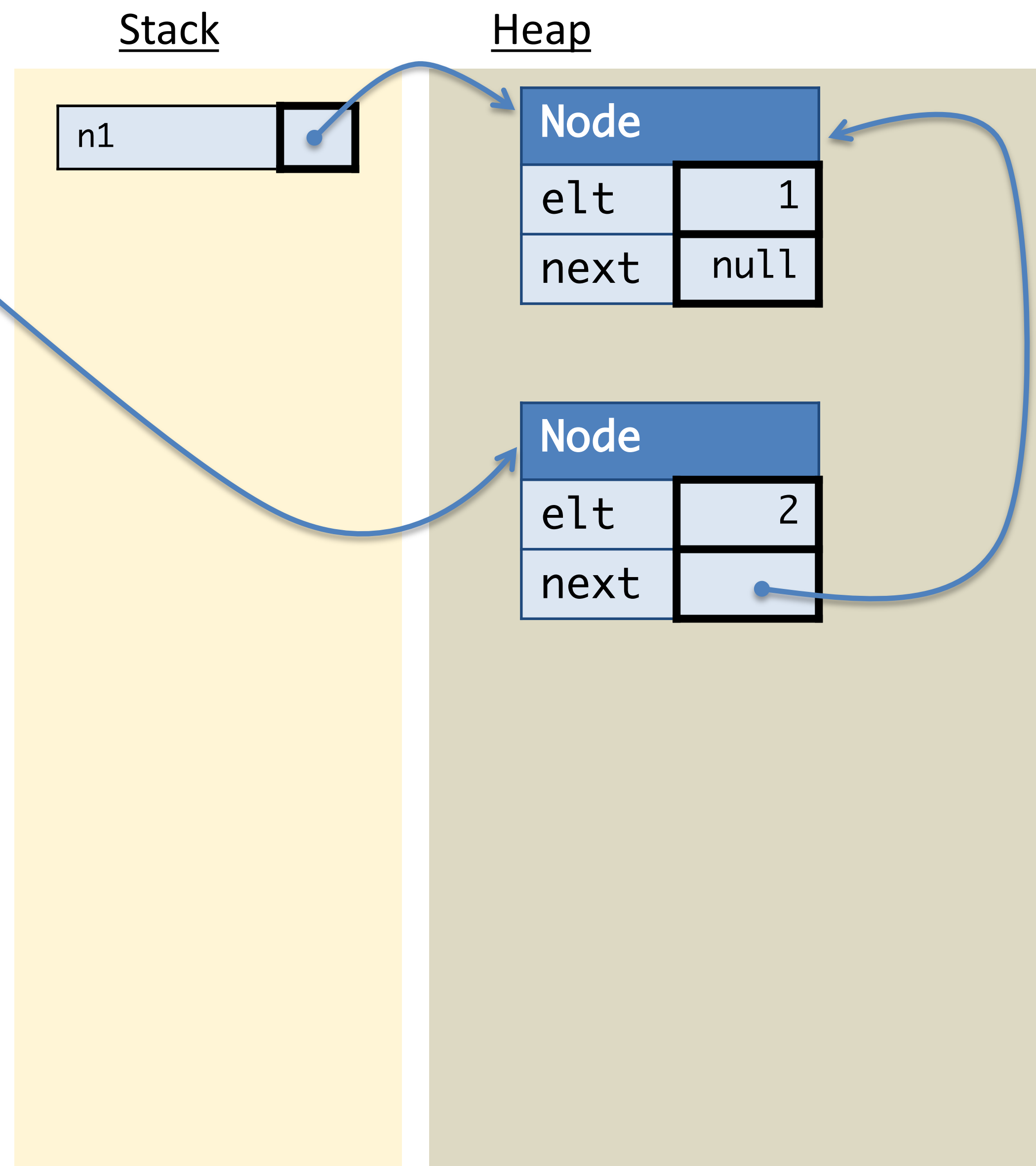
Workspace

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



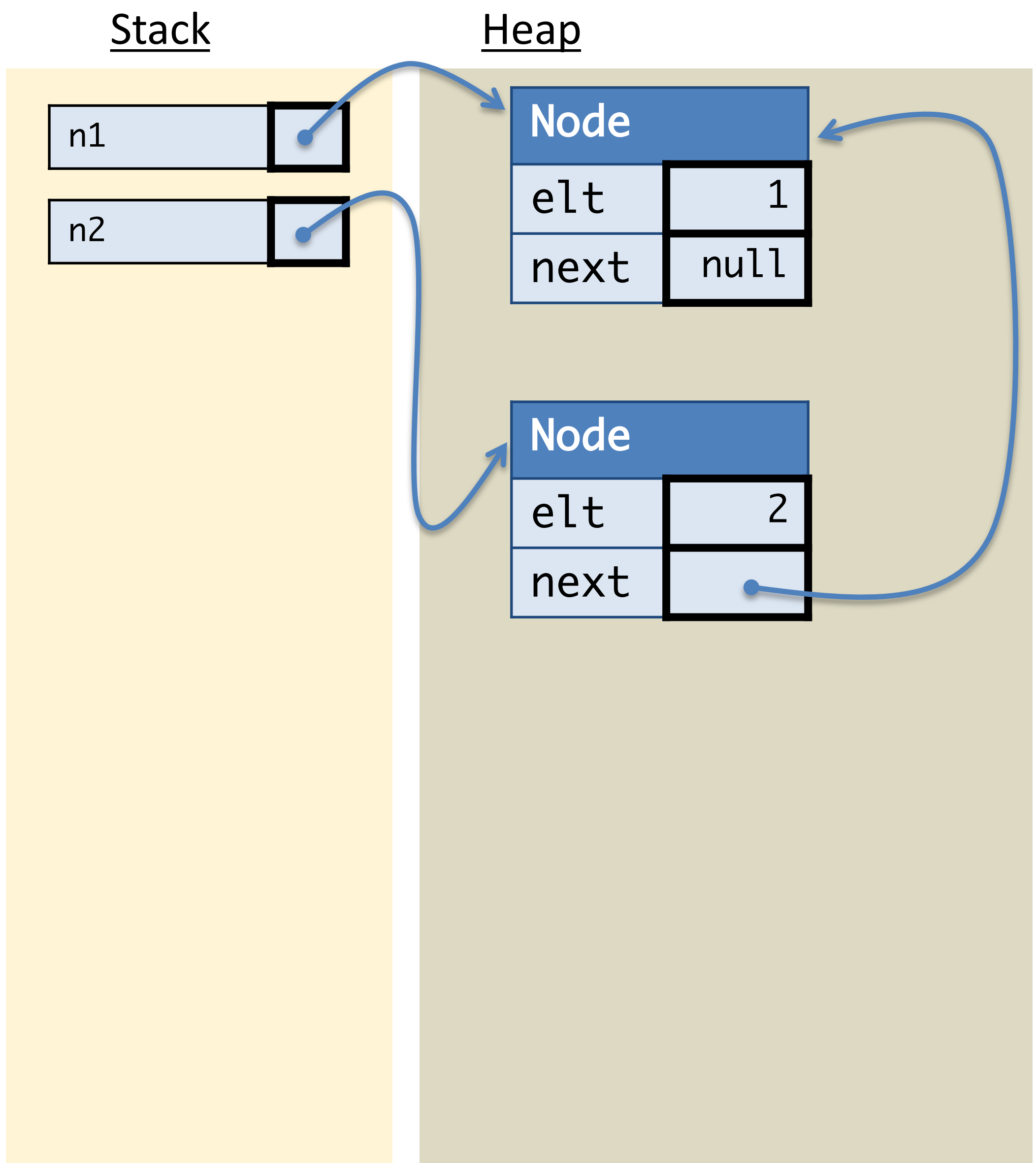
Workspace

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



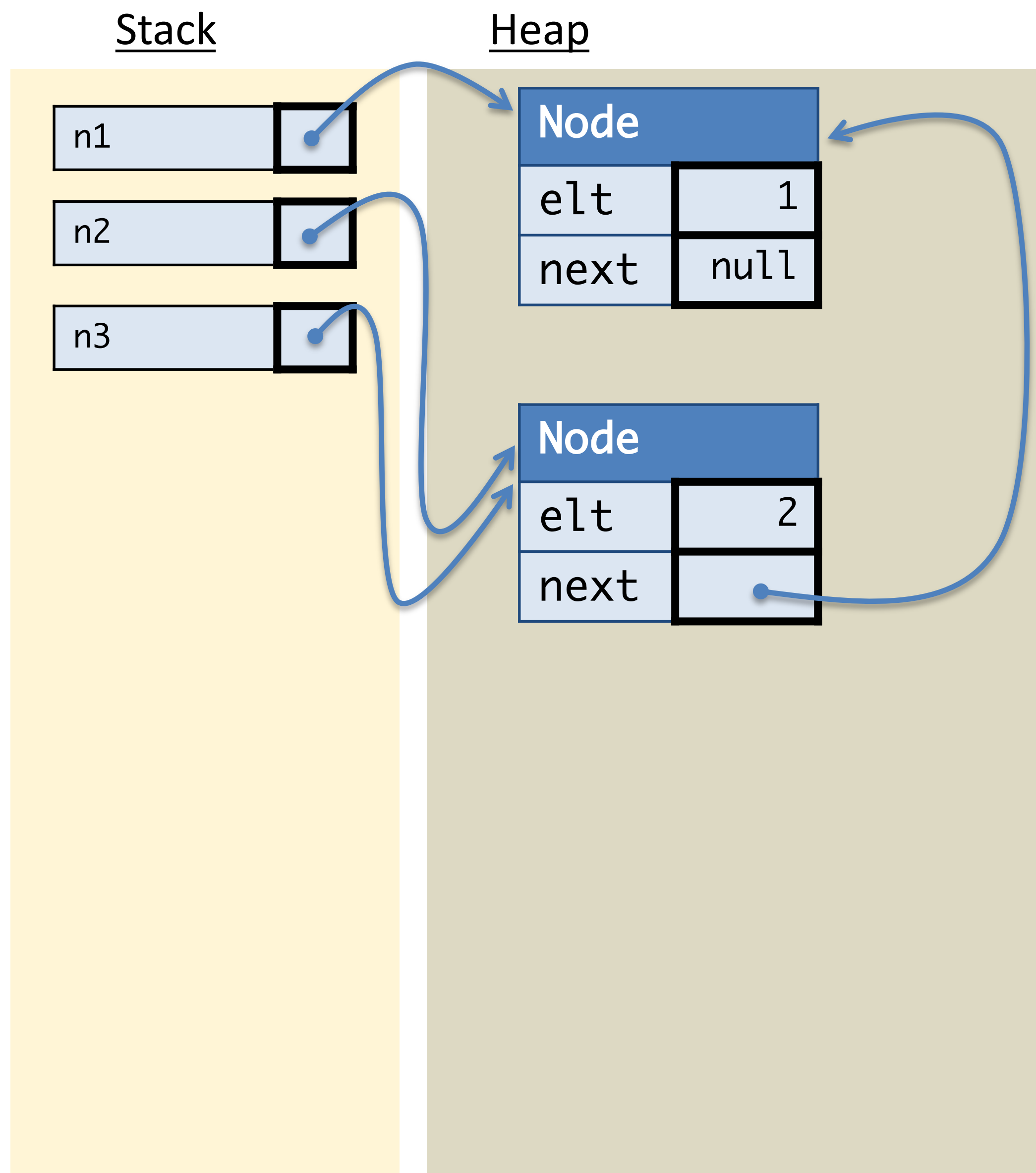
Workspace

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



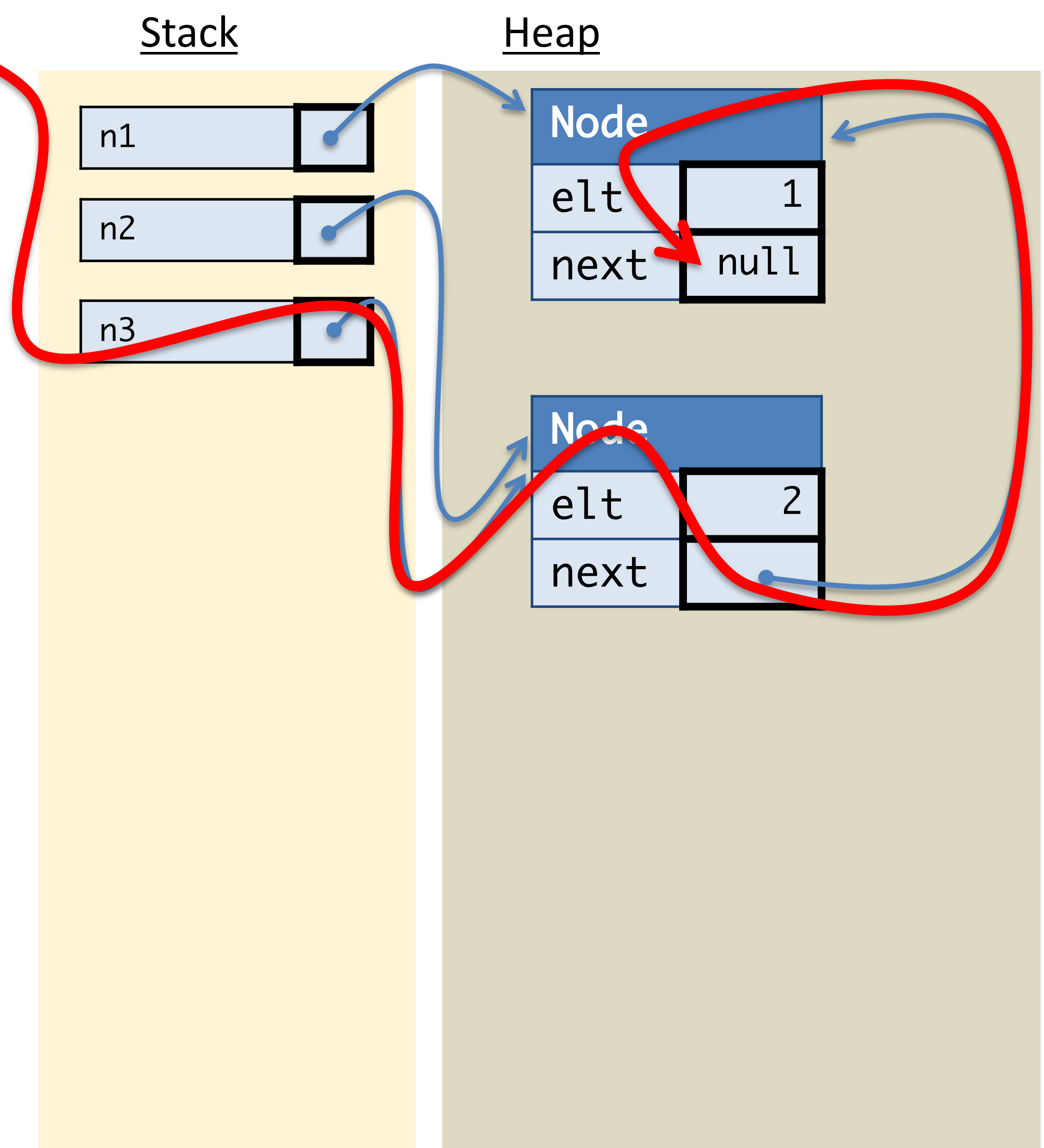
Workspace

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



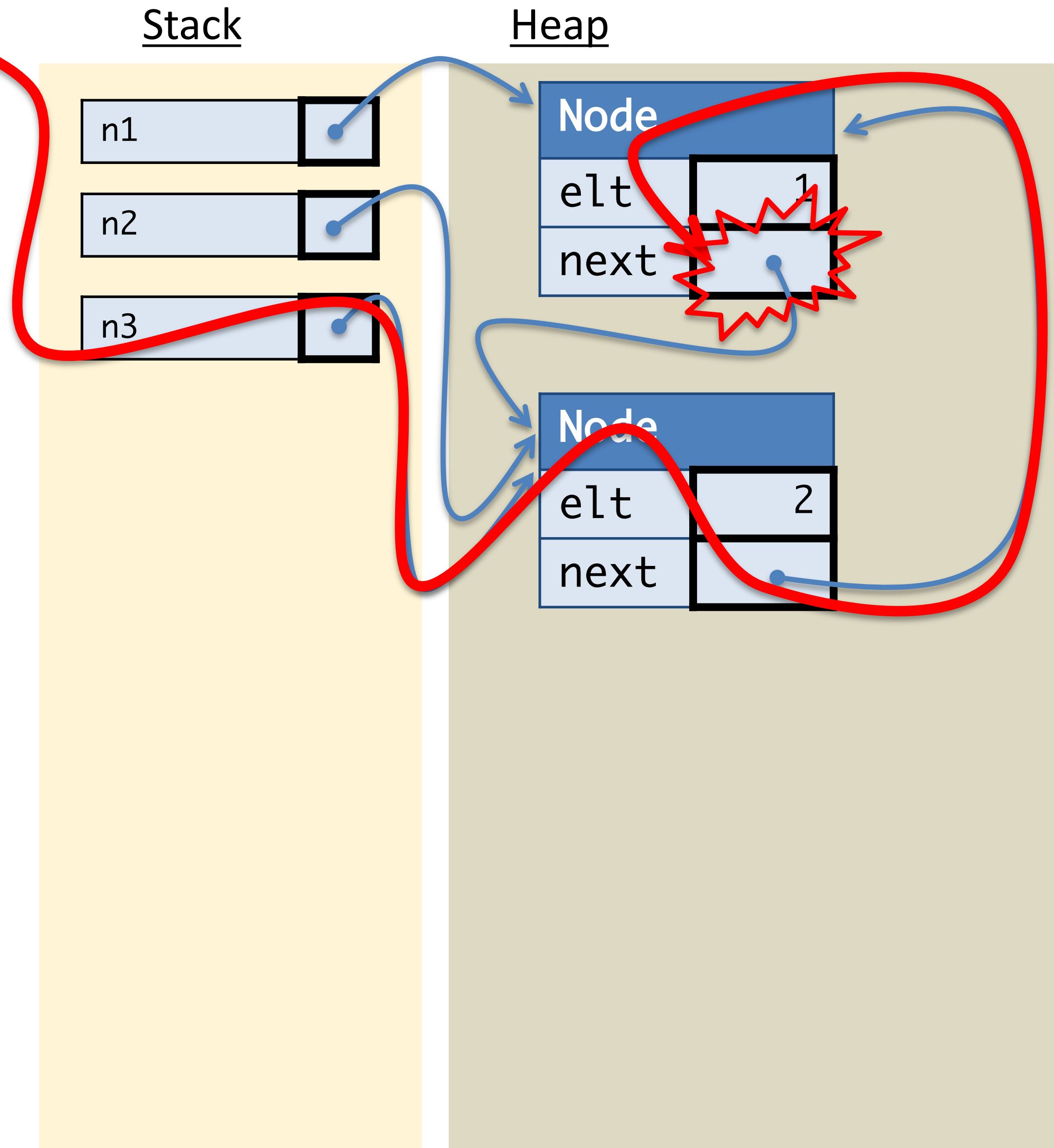
Workspace

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



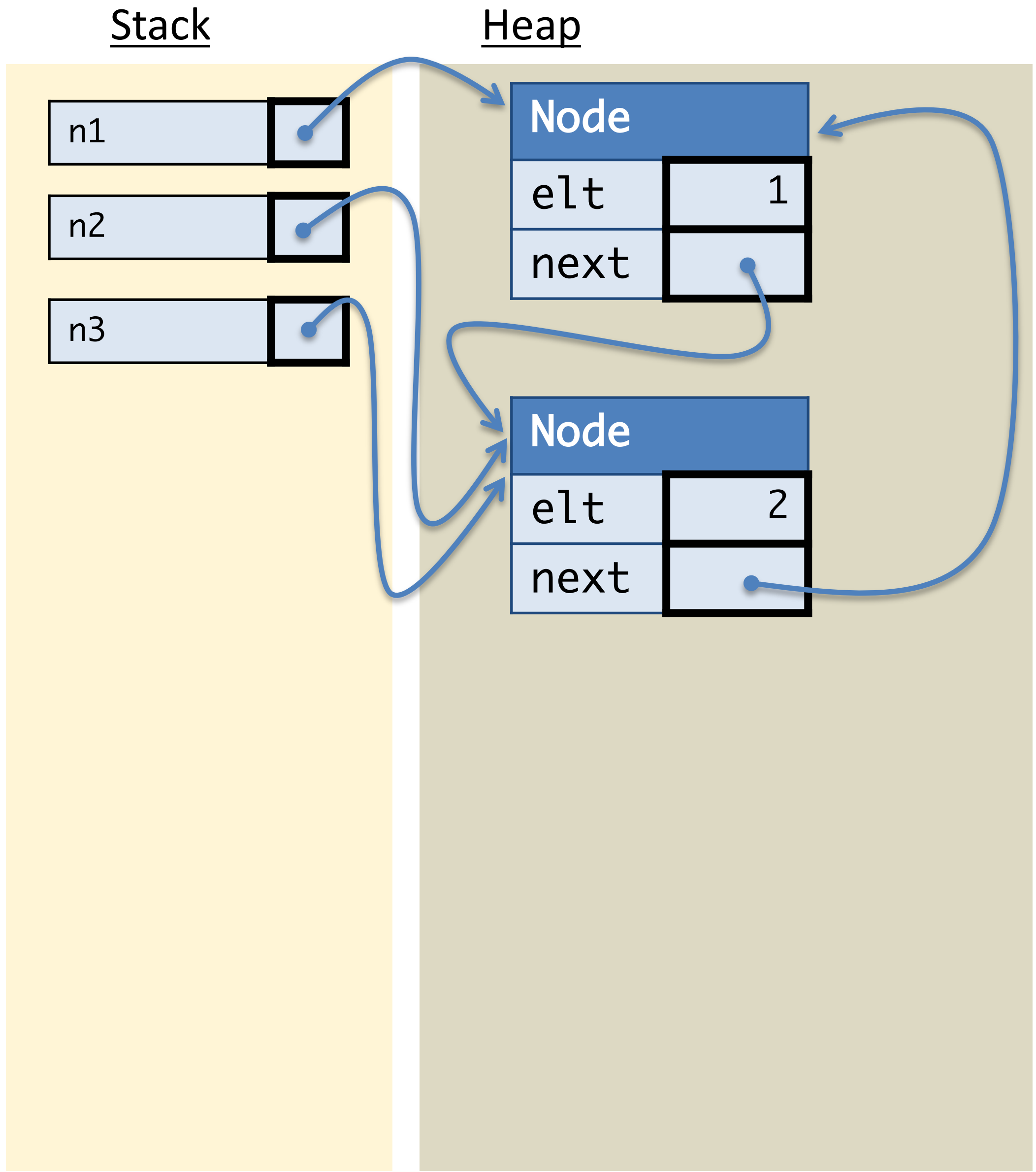
Workspace

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



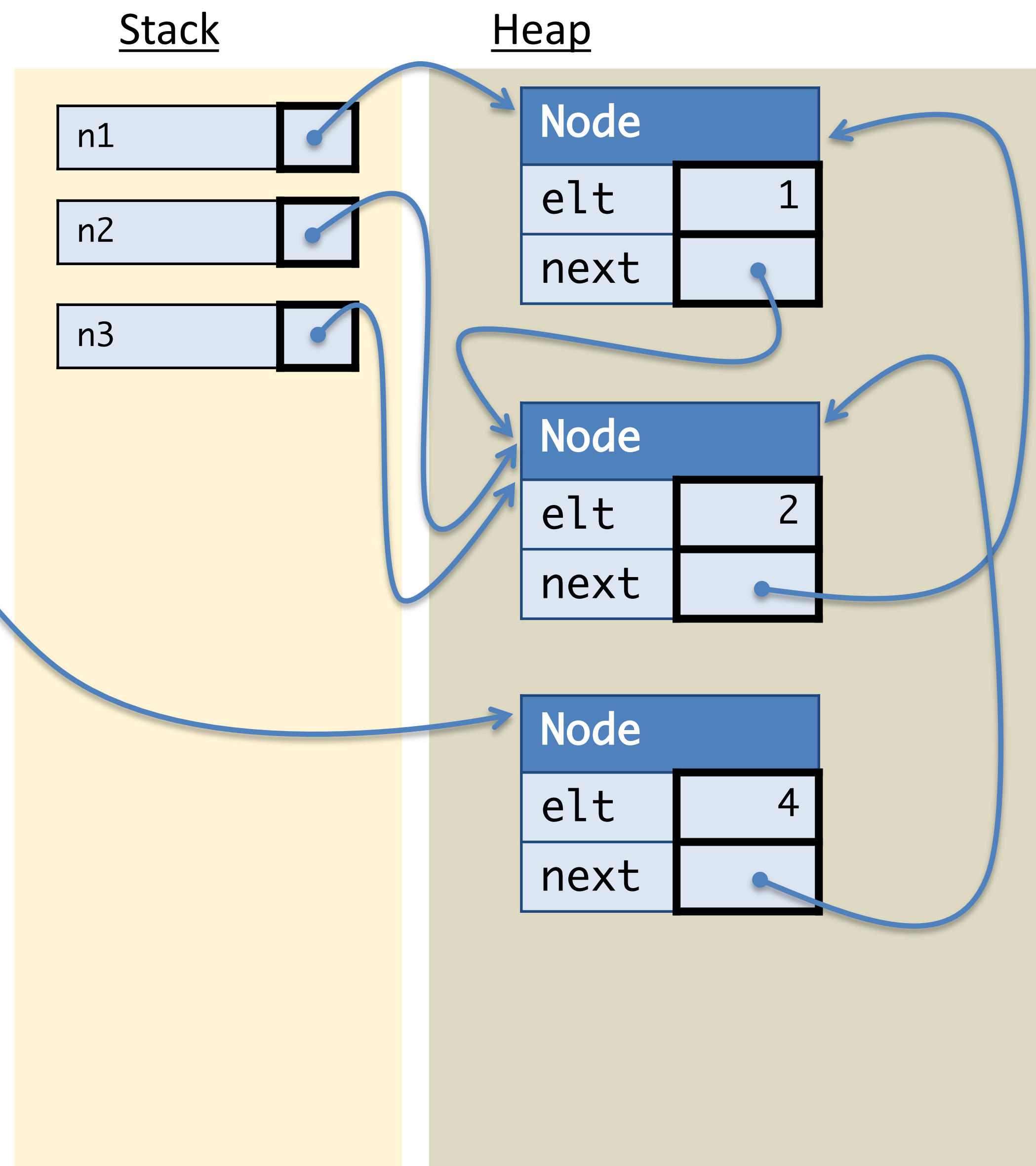
Workspace

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



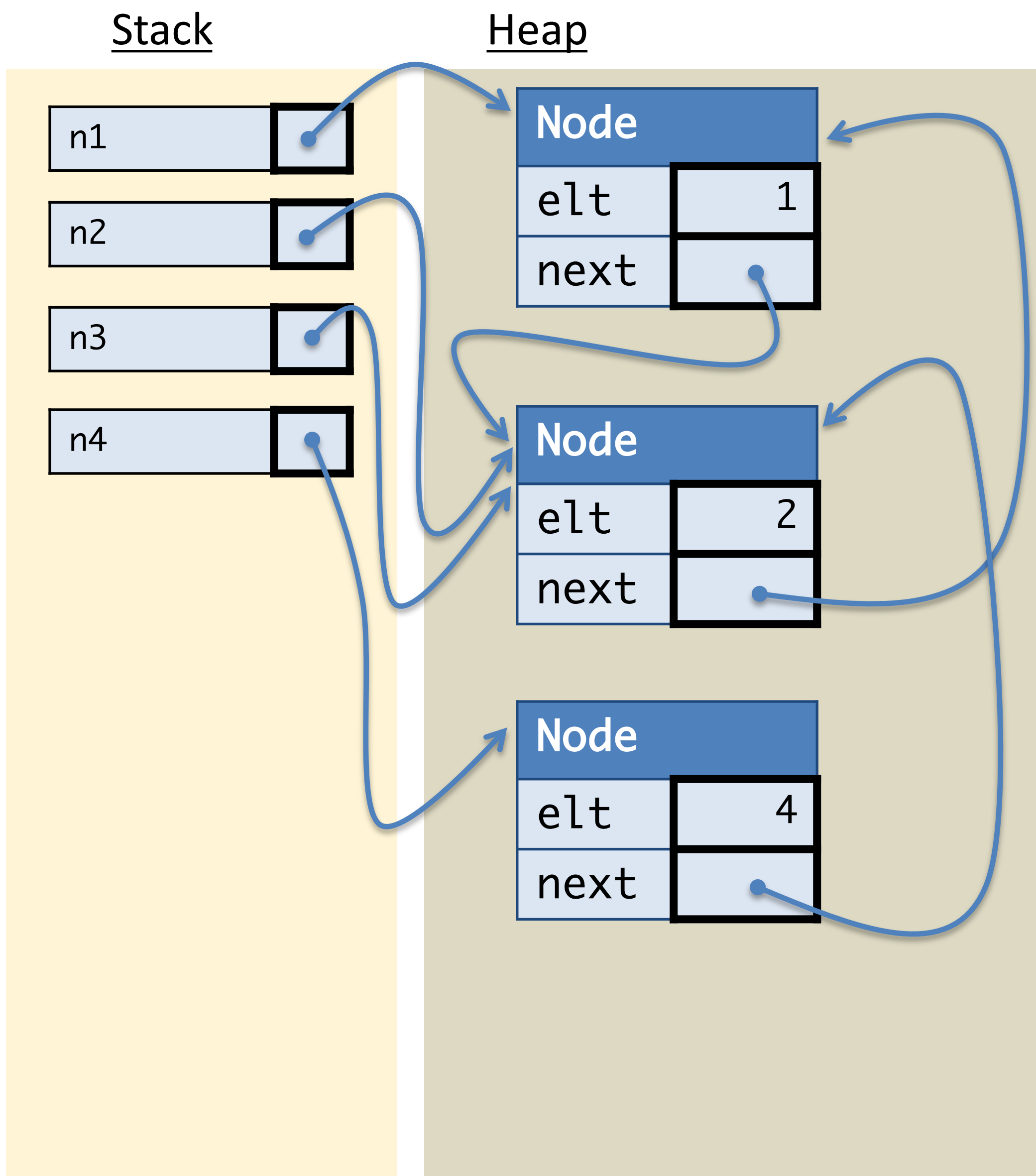
Workspace

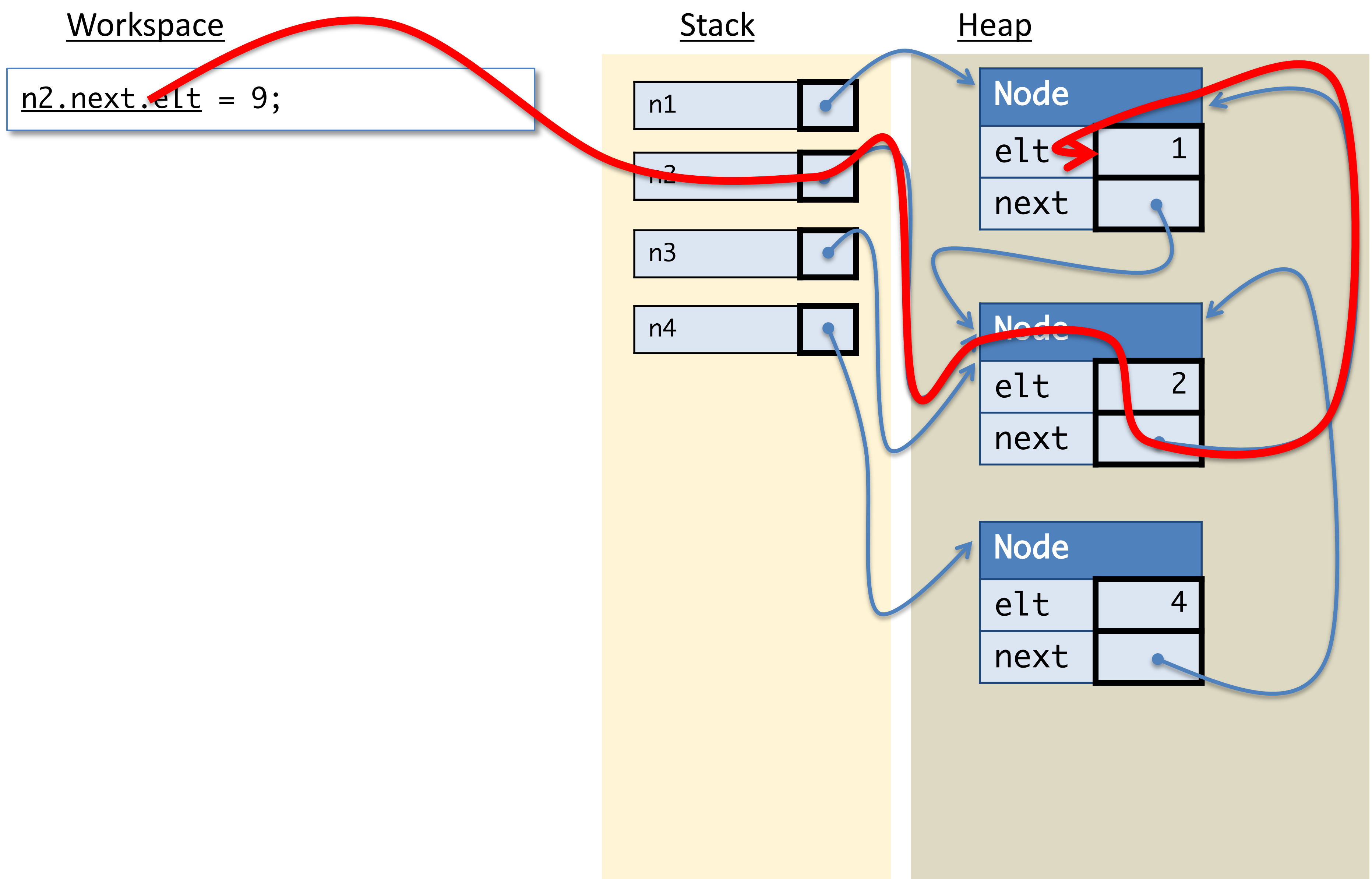
```
Node n4 = ;
n2.next.elt = 9;
```

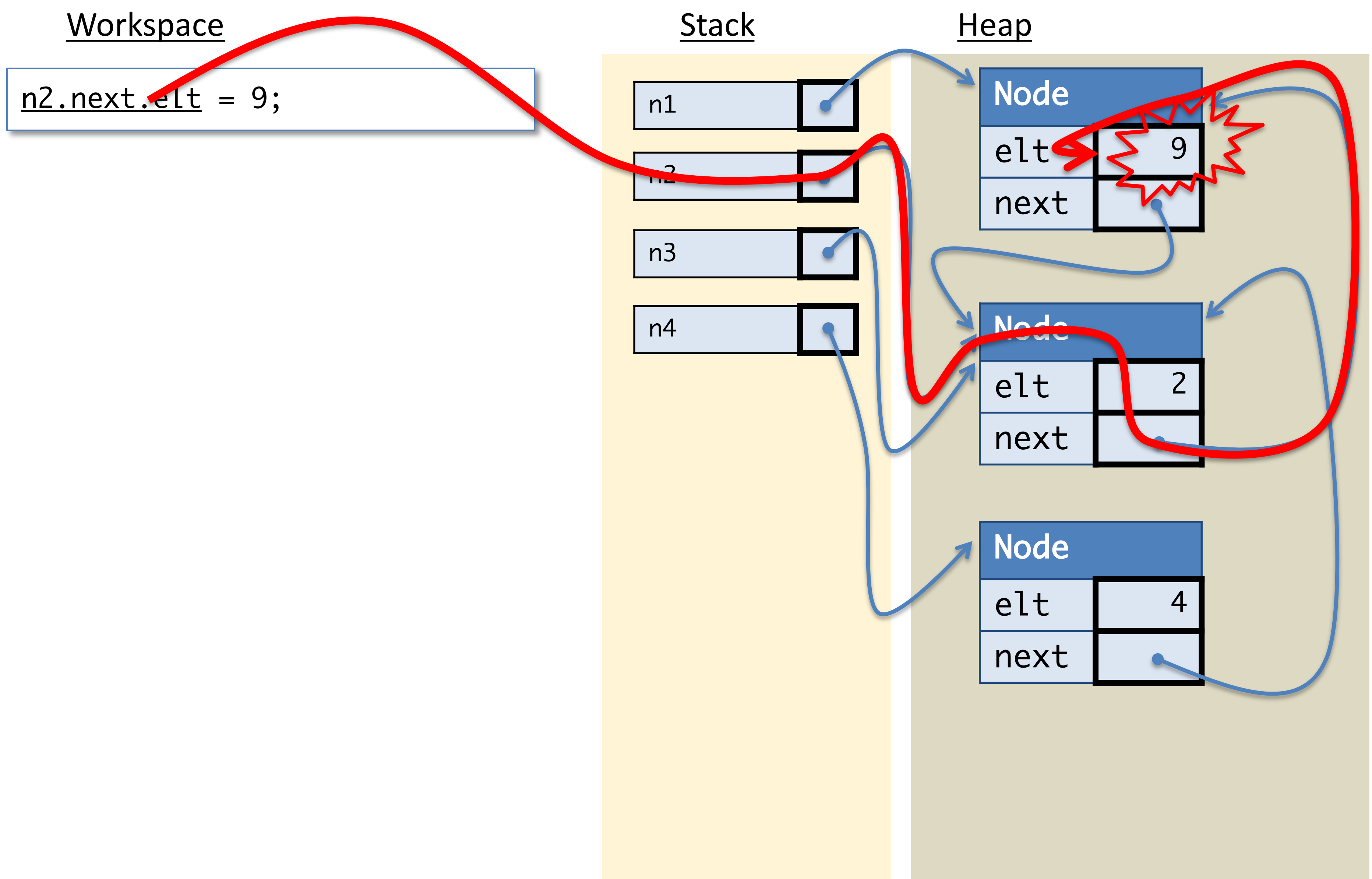


Workspace

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




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


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

OO

Subtypes

Quick Review:
Java Types and Interfaces

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 {
  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 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

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

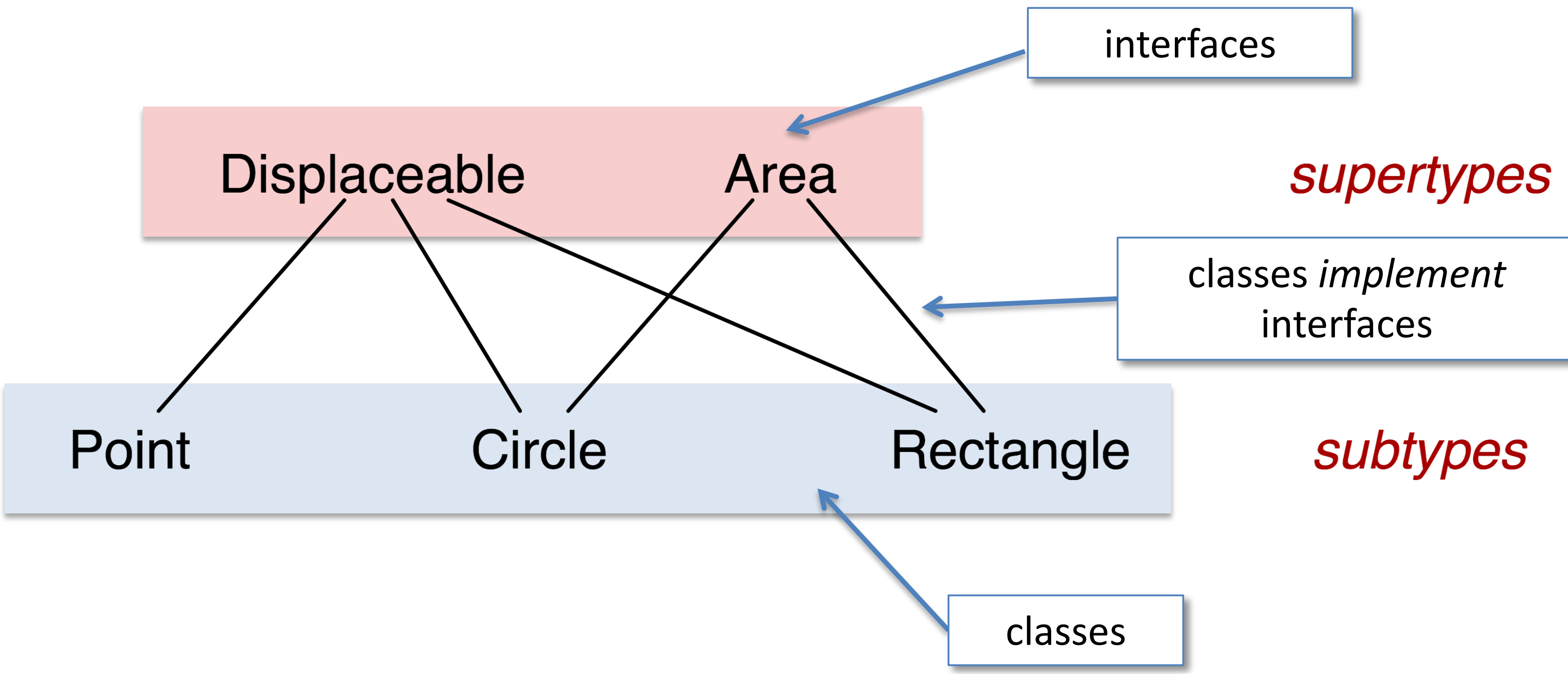
Subtyping

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

Extension

Interface Extension – An interface that *extends* another interface declares a subtype

Class Extension – A class that *extends* another class declares a subtype

Interface Extension

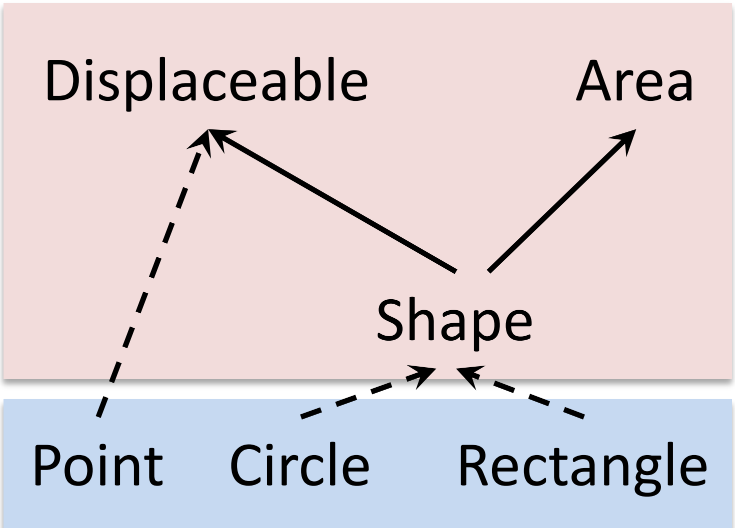
- Build richer interface hierarchies by *extending* existing interfaces.

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

public interface Area {
  double getArea();
}

public interface Shape extends Displaceable, Area {
    Rectangle getBoundingBox();
}
```

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

Note the "extends" keyword.

Interface Hierarchy




```
class Point implements Displaceable {
  … // omitted
}
class Circle implements Shape {
  … // omitted
}
class Rectangle implements Shape {
  … // omitted
}
```

- Shape is a *subtype* of both Displaceable and Area.
- Circle and Rectangle are both subtypes of Shape; by *transitivity*, both are also subtypes of Displaceable and Area.
- Note that one interface may extend *several* others.
  - Interfaces do not necessarily form a tree, but the interface hierarchy has no cycles.

Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only *one* other class.

- The extending class *inherits* all of the fields and methods of its *superclass*, and may include additional fields or methods.
  - This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).

- Design Tip: Class extension should *never* be used when “is a” does not relate the subtype to the supertype.

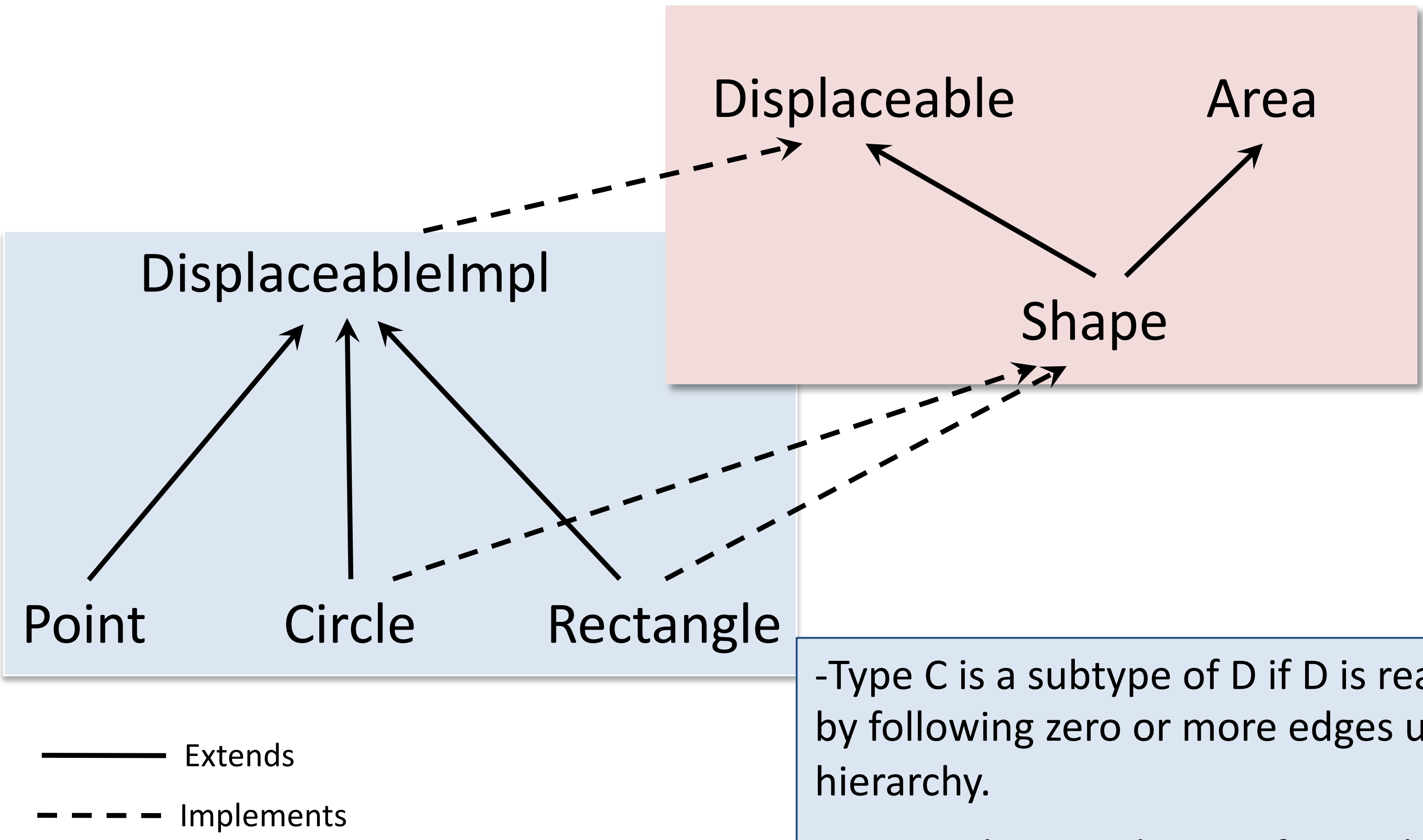
Simple Inheritance

- In *simple inheritance*, the subclass only *adds* new fields or methods.
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

Class Extension: Inheritance

```
public class DisplaceableImpl implements Displaceable {
   private int x; private int y;
   public DisplaceableImpl(int x, int y) { … }
   public int getX() { return x;}
   public int getY() { return y; }
   public void move(int dx, int dy) { x += dx; y += dy; }
}
public class Circle extends DisplaceableImpl
                                     implements Shape {
   private int radius;
   public Circle(Point pt, int radius) {
     super(pt.getX(),pt.getY());
     this.radius = radius;
   }
   public double getArea() { … }
   public Rectangle getBoundingBox() { … }
}
```

Subtyping with Inheritance



-Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.

- e.g. Circle is a subtype of Area, but Point is not

Example of Simple Inheritance

See: Shapes.zip

Inheritance: Constructors

- Contructors are *not* inherited
  - Instead, each subclass constructor should invoke a constructor of the superclass using the keyword super
  - Super *must* be the first line of the subclass constructor
    - if the parent class constructor takes no arguments, it is OK to omit the explicit call to super (it will be supplied automatically)

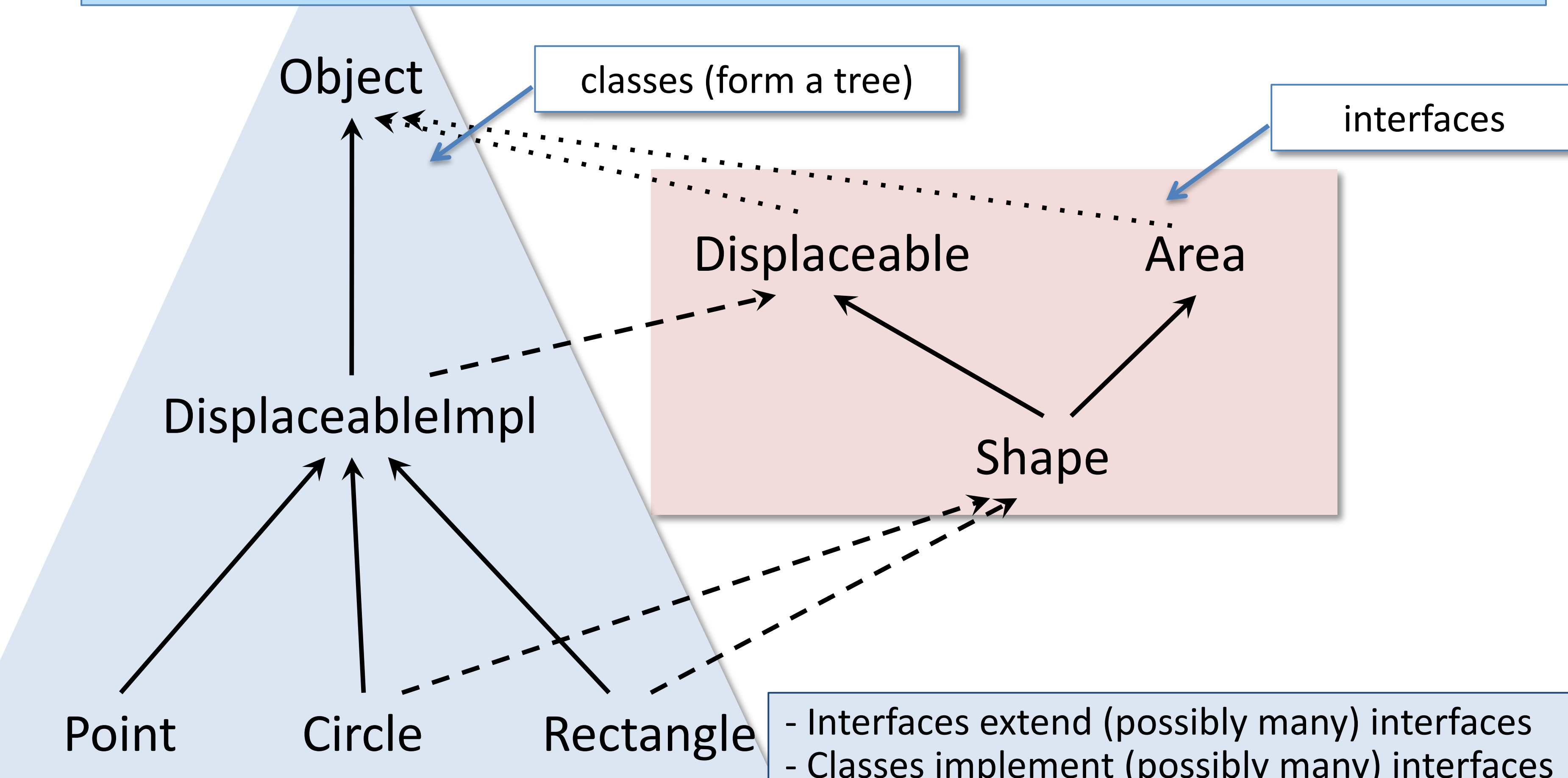
```
public Circle(Point pt, int radius) {
     super(pt.getX(),pt.getY());
     this.radius = radius;
   }
```

Class Object

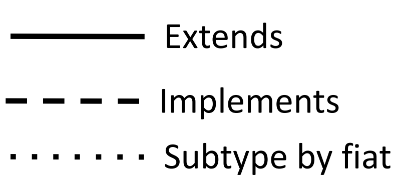
```
public class Object {
  boolean equals(Object o) {
    … // test for equality
  }
  String toString() {
    … // return a string representation
  }
  …  // other methods omitted
}
```

- Object is the root of the class tree
  - Classes with no “extends” clause *implicitly* extend Object
  - Arrays also implement the methods of Object
  - This class provides methods useful for *all* objects to support
- Object is the top (i.e., “most super”) type in the subtyping hierarchy

Recap



- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes "by fiat" of Object



Other forms of inheritance

- Java has other features related to inheritance (some of which we will discuss later in the course):
  - A subclass might *override* (re-implement) a method already found in the superclass.
  - A class might be *abstract* – i.e. it does not provide implementations for all of its methods (its subclasses must provide them instead)
- These features are tricky to use properly, and the need for them arises only in somewhat special cases
  - Designing complex, reusable libraries
  - Special methods like `equals` and `toString`
- We recommend avoiding *all* forms of inheritance (even "simple inheritance") whenever possible: use interfaces and composition instead

Especially: Avoid method overriding except in a few special cases