Programming Languages and Techniques (CIS120)

Lecture 24

Oct 28, 2015

Interfaces and Subtyping

Announcements

- HW 06 due Thursday (tomorrow) at midnight

- Midterm II, in class, a week from Friday (Nov 6th)
  - Details on Friday

The Java Abstract Stack Machine

Objects, Arrays, and Static Methods

What does the following program print?

1 – 9

or 0 for "NullPointerException"

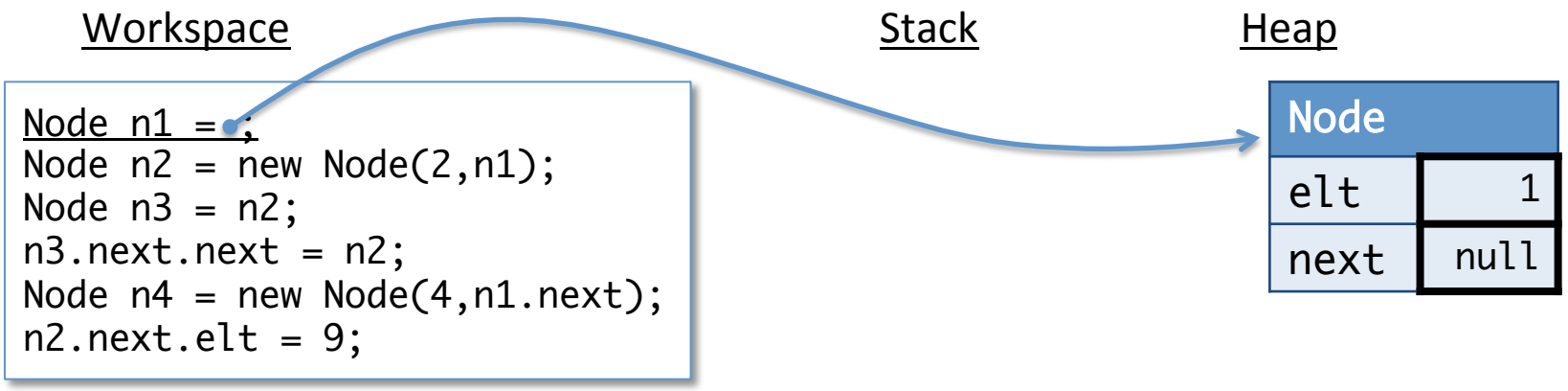
```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}
public class Test {
  public static void main(String[] args) {
    Node n1 = new Node(1,null);
    Node n2 = new Node(2,n1);
    Node n3 = n2;
    n3.next.next = n2;
   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

}
```

Workspace

Stack

Heap

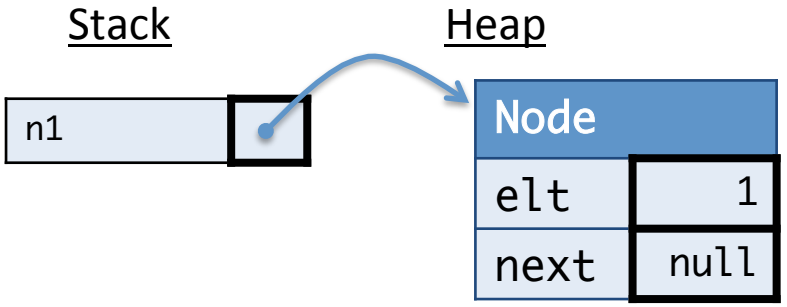
```
Node n1 = new Node(1,null);
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

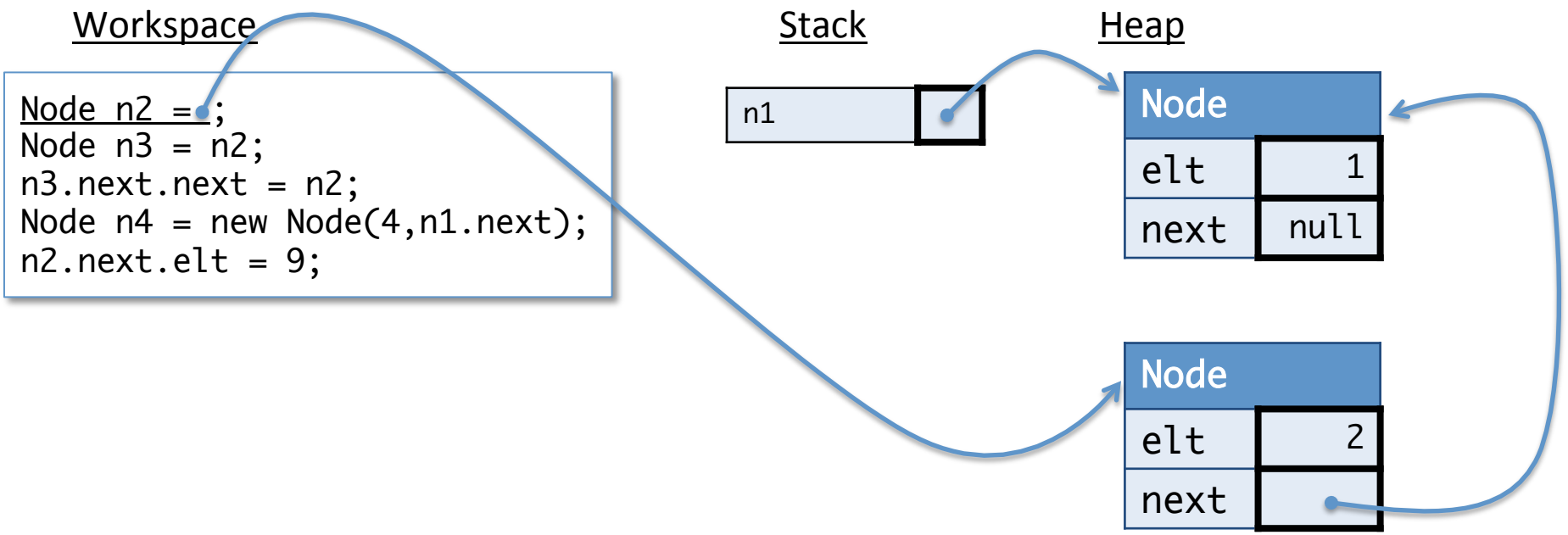


Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

Workspace

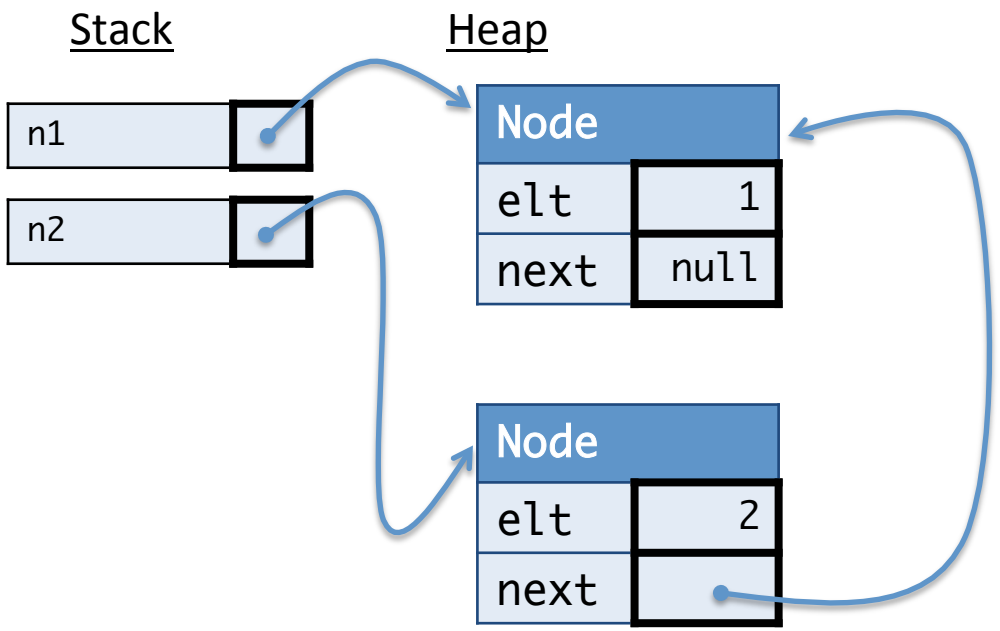
```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```




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

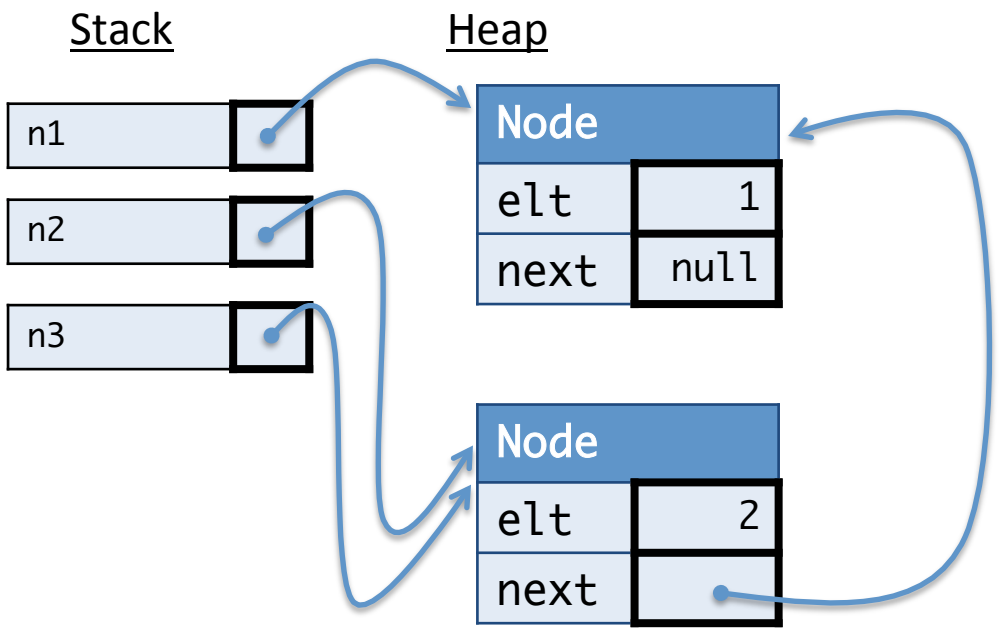
Workspace

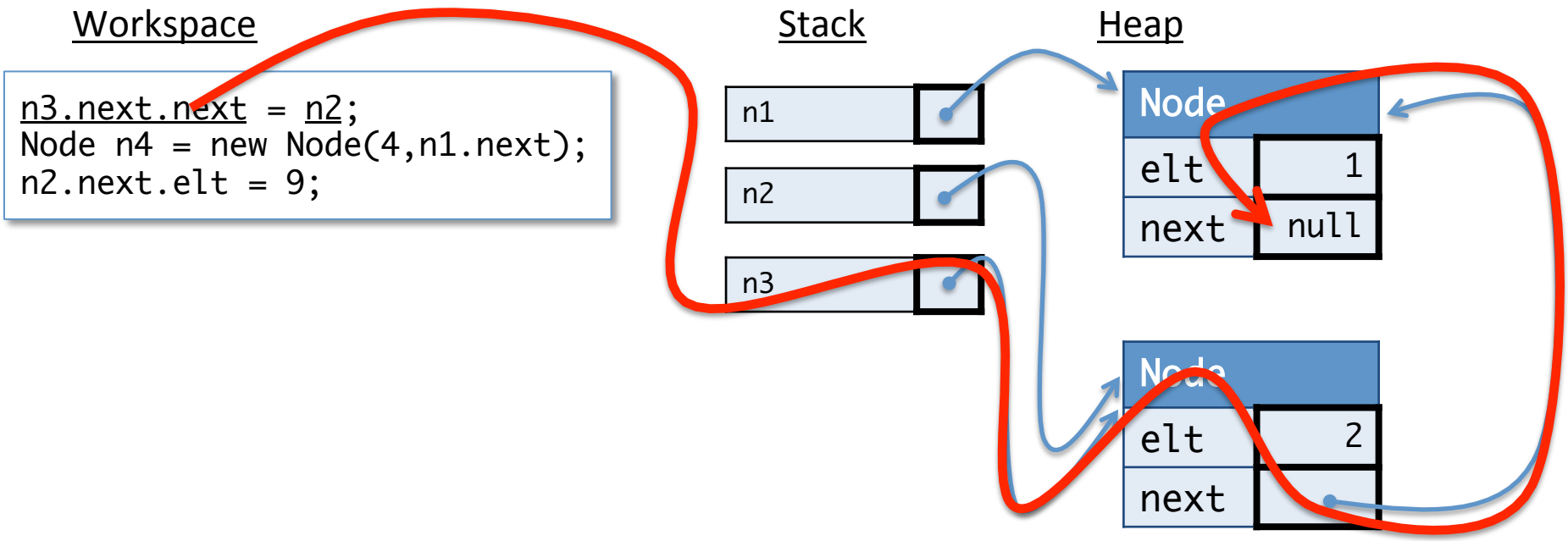
```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

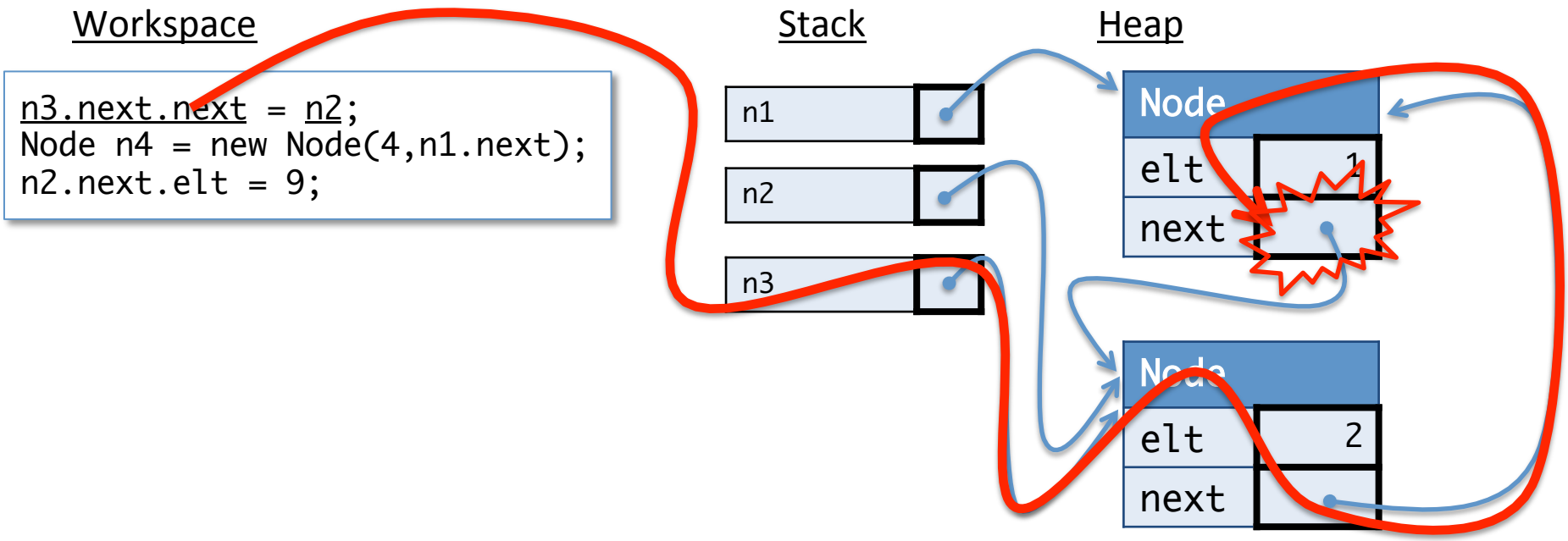


Workspace

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

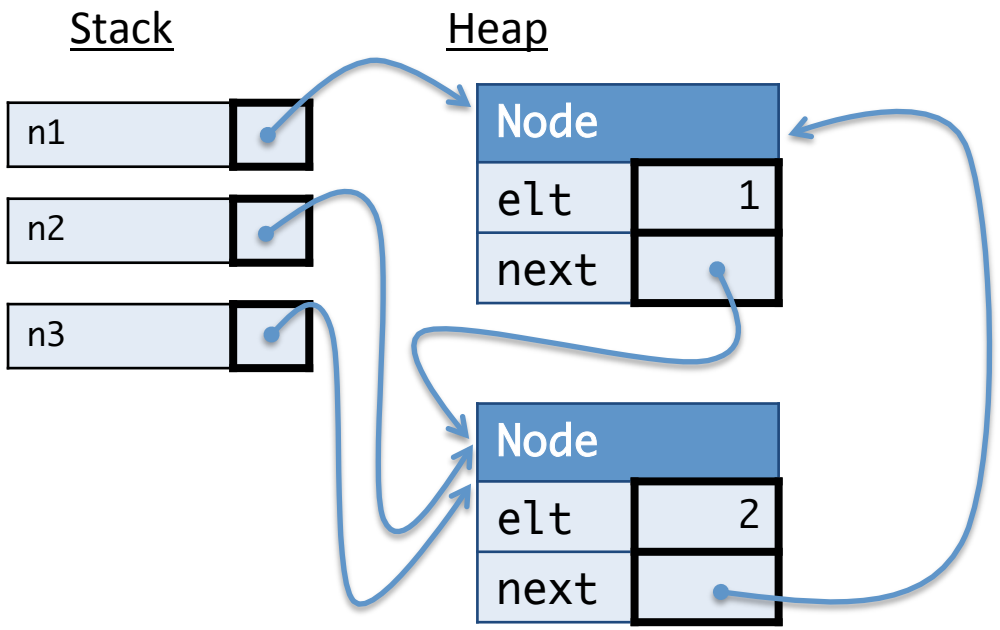


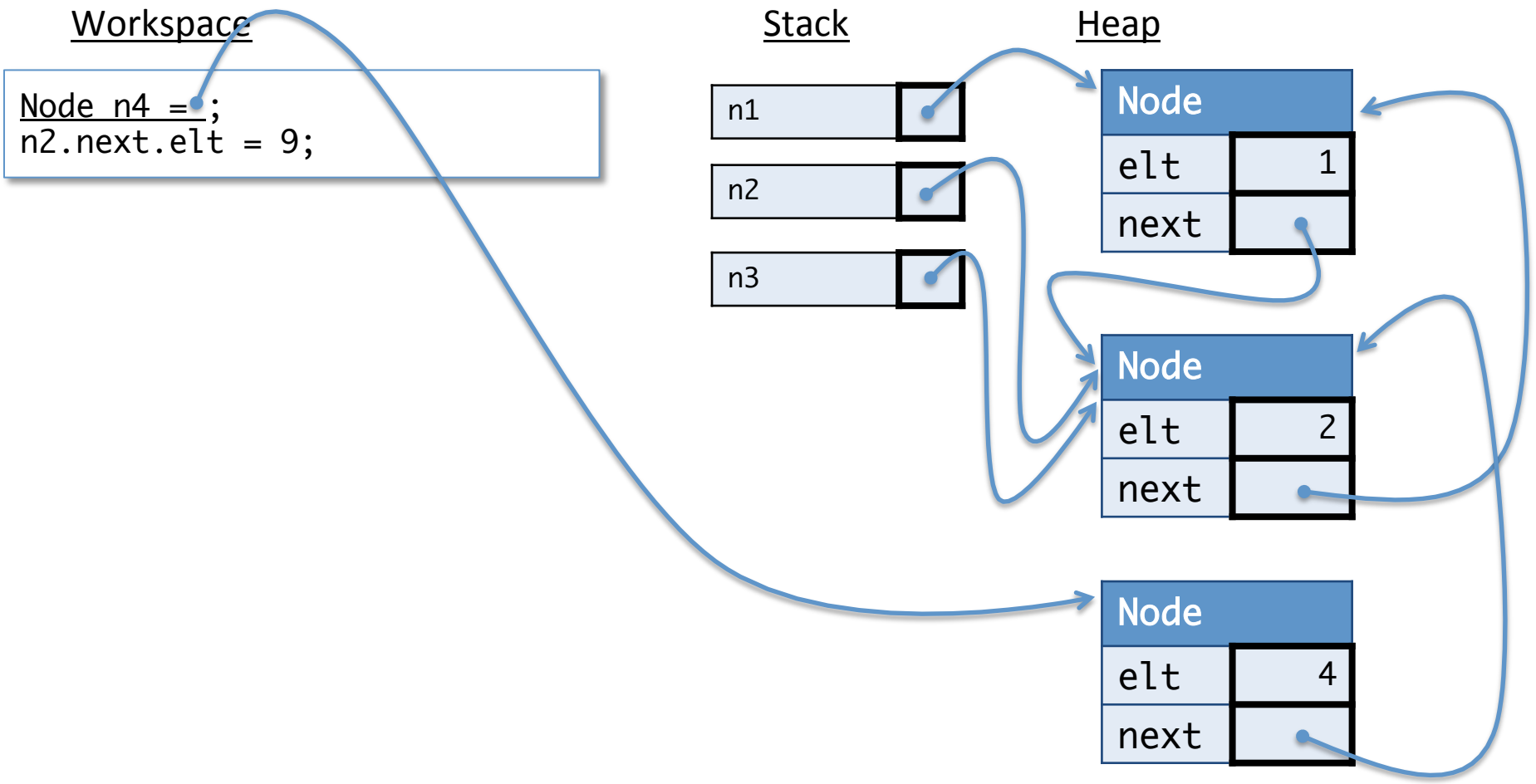

Workspace
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
null
Node
elt
2
next


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

Workspace

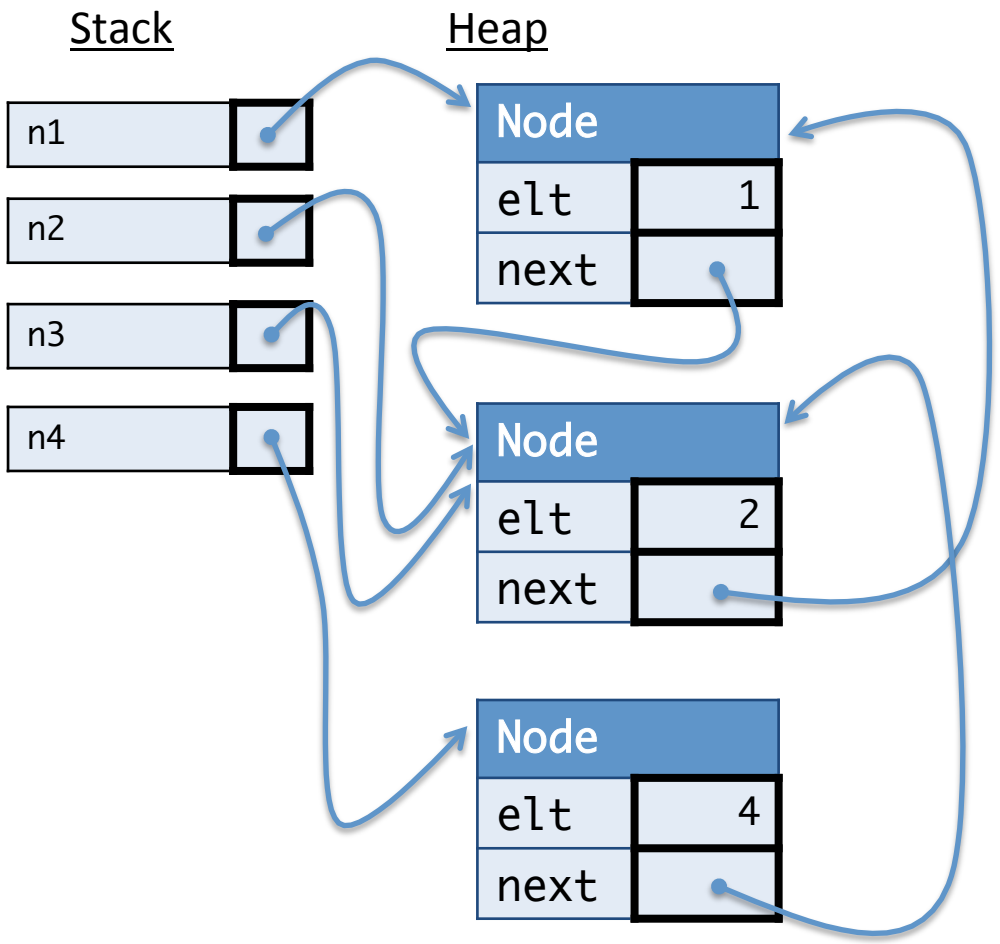
```
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

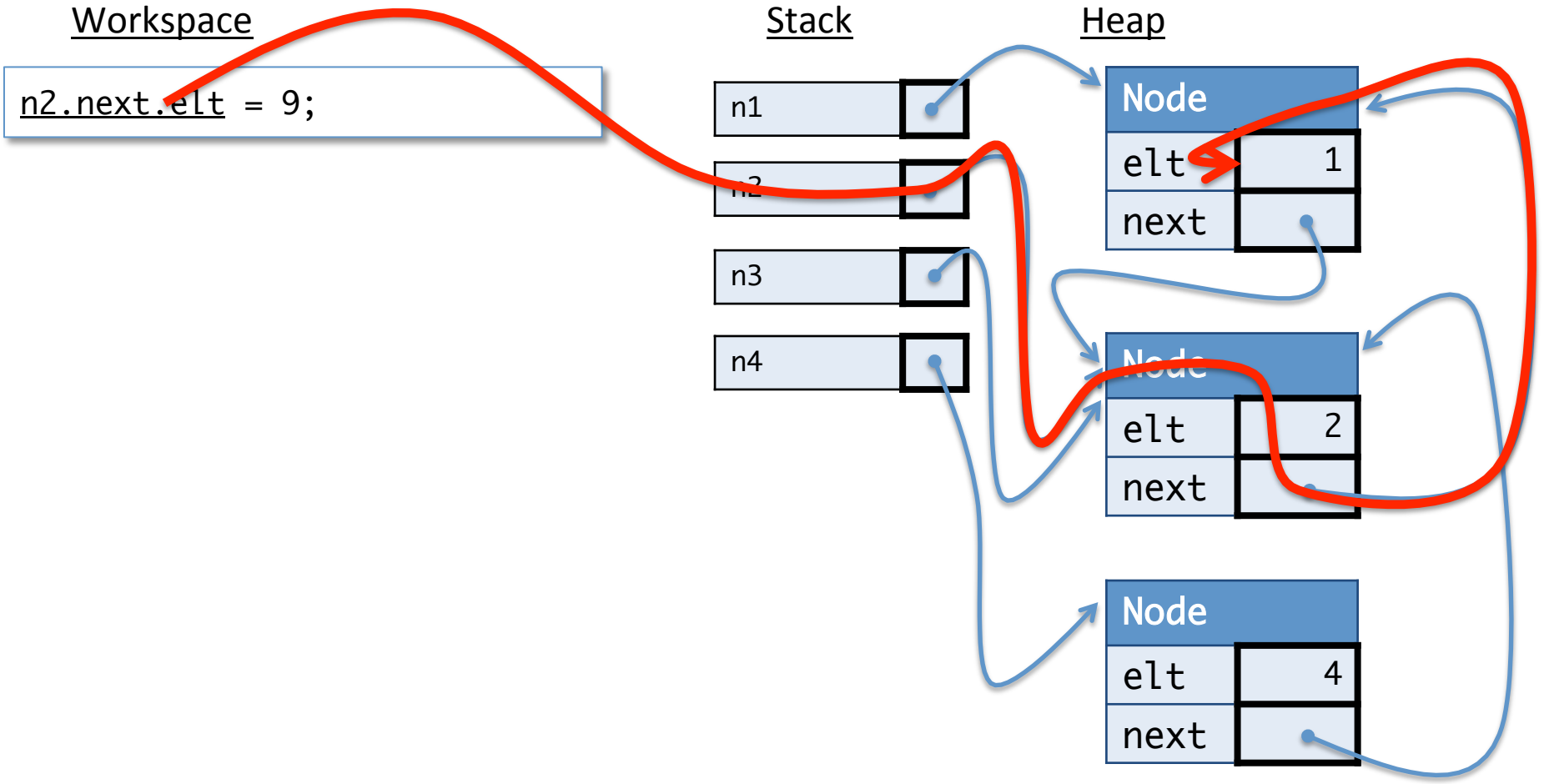


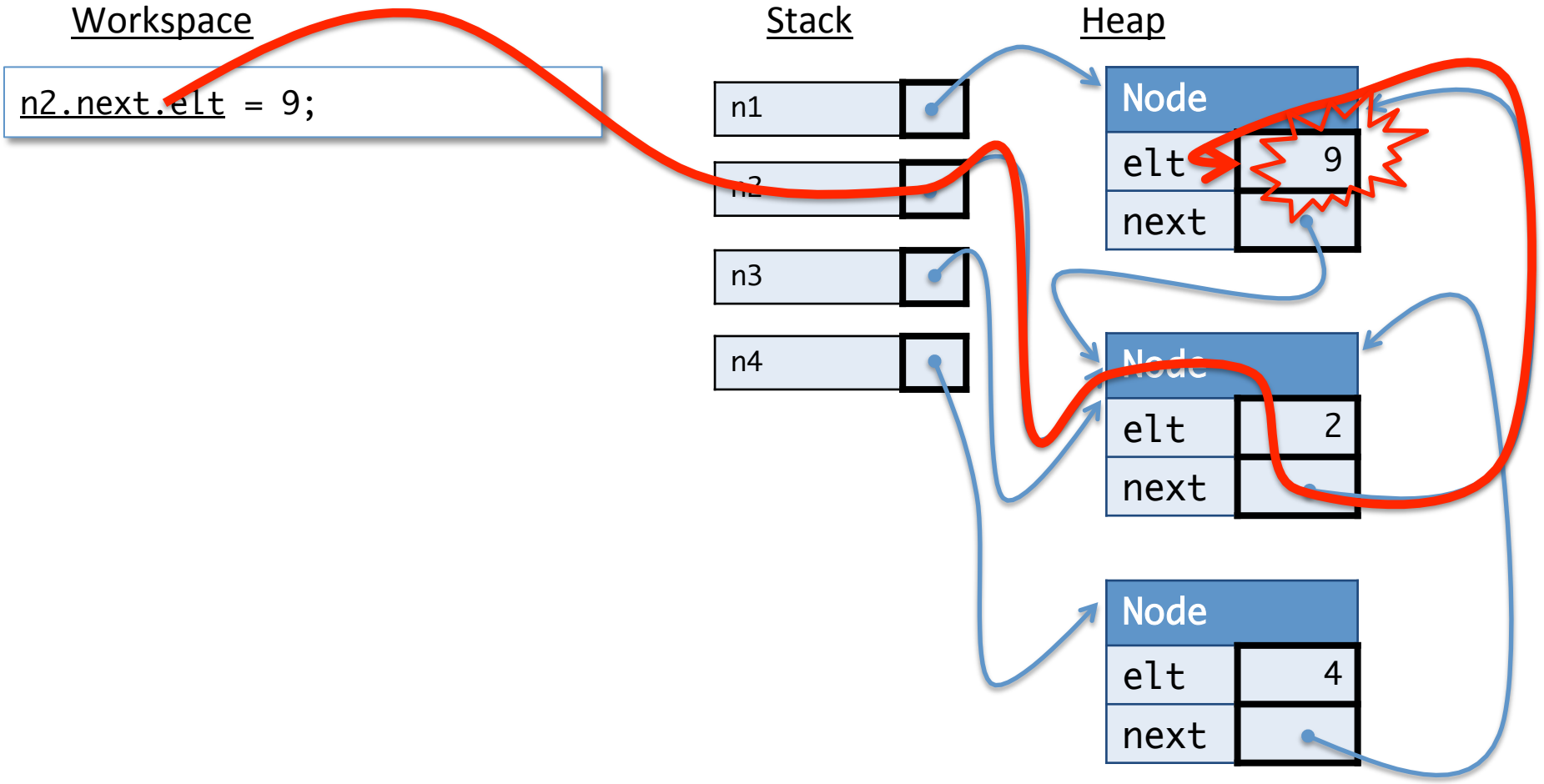

Workspace
Node n4 =;
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

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

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....

Interfaces

Working with objects abstractly

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

```
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

```
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(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

Another implementation

```
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

Interfaces are types

- Can declare variables of interface type

```
void m(Displaceable d) { … }
```

- Can call method with any Displaceable argument…

```
obj.m(new Point(3,4));
obj.m(new ColoredPoint(1,2,Color.Black));
```

- … but m can only operate on d according to the interface

```
d.move(-1,1);
…
… d.getX() …          ⇒ 0.0
… d.getY() …          ⇒ 3.0
```

Using interface types

- Interface variables can refer (during execution) to objects of any class implementing the interface
- Point, Circle, and ColoredPoint are all *subtypes* of Displaceable

```
Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2,3), 1);
d2 = new ColoredPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
d2.move(-2,0);
…
… d0.getX() …      ⇒ -1.0
… d1.getX() …      ⇒  0.0
… d2.getX() …      ⇒ -3.0
```

Class that created the object value determines what move function is called.

Abstraction

- The interface gives us a single name for all the possible kinds of "moveable things." This allows us to write code that manipulates arbitrary `Displaceable` objects, without caring whether it's dealing with points or circles.

```
class DoStuff {
  public void moveItALot (Displaceable s) {
    s.move(3,3);
    s.move(100,1000);
    s.move(1000,234651);
  }

  public void dostuff () {
    Displaceable s1 = new Point(5,5);
    Displaceable s2 = new Circle(new Point(0,0),100);
    moveItALot(s1);
    moveItALot(s2);
  }
}
```

Multiple interfaces

- An interface represents a point of view

  …but there can be multiple valid points of view

- Example: Geometric objects
  - All can move (all are Displaceable)
  - Some have Color (are Colored)

Colored interface

- Contract for objects that that have a color
  - Circles and Points don't implement Colored
  - ColoredPoints do

```
public interface Colored {
    public Color getColor();
}
```

ColoredPoints

```
public class ColoredPoint
 implements Displaceable, Colored {

  Point center;
  private Color color;
  public Color getColor() {
    return color;
  }

  …
}
```

Static vs. Dynamic Types

Subtyping

Definition:
Type A is a *subtype* of type B if A offers the same public methods that B can does.

- Type B is called the *supertype* of A.
- Intuitively: an A object can do anything that a B object can
- Note: A may provide *more* public methods

Explicit Subytping

- Java requires subtypes to be declared *explicitly* via keywords `implements` and `extends`
  - there is no subtyping by "coincidence" (i.e. just because the public method names happen to be the same)

- **Example**: A class that implements an interface is a subtype of the interface:

```
interface Displaceable { … }

public class Circle implements Displaceable {
    …
}
```

Subtyping and Variables

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

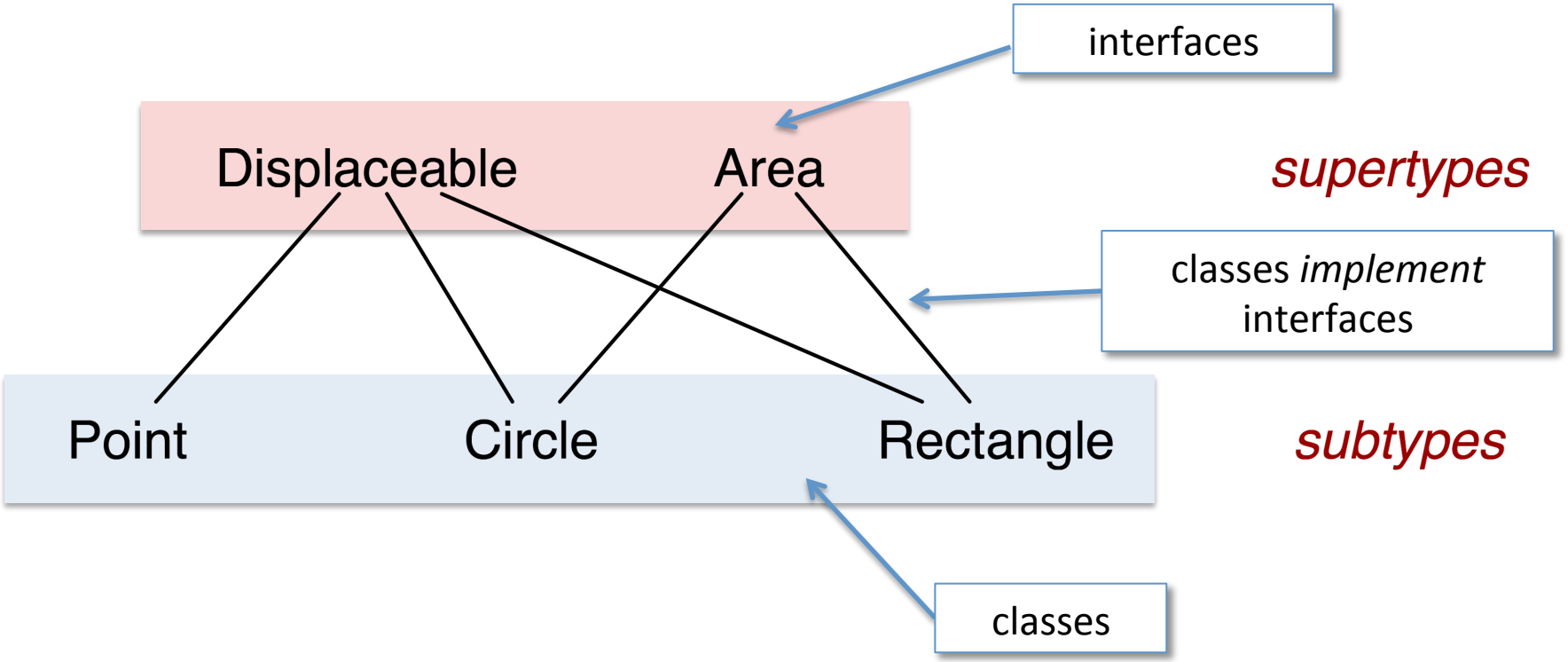
```
Area a = new Circle(1, new Point(2,3));
```

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

```
static 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

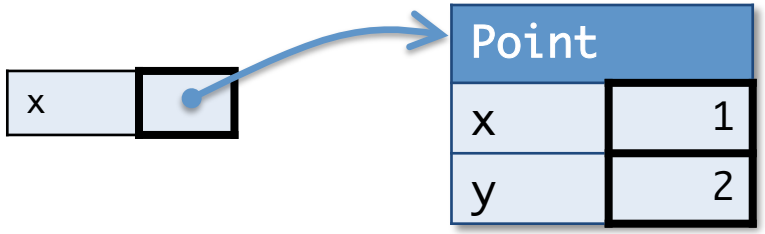
"Static" types vs. "Dynamic" classes

- 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

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?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 dynamic class of s1 when execution reaches A?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 s2 on line B?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 type should we declare for x (in blank D)?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 dynamic class of x?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 type should we declare for y (in blank E)?

1. Rectangle
2. Circle
3. Area
4. none of the above

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 dynamic class of y?

1. Rectangle
2. Circle
3. Area
4. none of the above

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
```

Is the assignment on line F well typed?

1. yes
2. no

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
```

Is the assignment on line G well typed?

1. yes
2. no

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
```

Is the assignment on line H well typed?

1. yes
2. no

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
```

Is the assignment on line I well typed?

1. yes
2. no

Extension

1. Interface extension
2. Class extension (Simple inheritance)