Programming Languages and Techniques (CIS1200)

Lecture 24

Java ASM, Subtyping and extension

Chapters 22 and 23

Announcements

- HW06: Pennstagram
  - Java array programming
  - Due *Tuesday* (tomorrow) at 11.59pm

Midterm 2 Logistics

- **Friday, March 28th, 2025**
  - ***During lecture:* 1:45-2:45PM**
  - If you have a conflict, send email to cis1200@seas.upenn.edu ASAP
- **Location**: Meyerson B1 (MEYH)
- **Coverage:** Chapters 1-24
- **Format:** 60 minutes; closed book, one handwritten, letter sized, single sided sheet of notes allowed.
- **Review Session:**
  Wednesday, March 26 from 7-9pm in Towne 100

The Java Abstract Stack Machine

Objects, Arrays, and Static Methods

Java Abstract Stack Machine

- Similar to OCaml Abstract Stack Machine
  - Workspace (currently executing code)
  - Stack (local variables, plus saved workspaces in method calls)
  - Heap (values of reference types: arrays and objects)

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

Java Primitive Values

The values of these data types fit into one machine "word" (i.e. 64 bits) and are stored directly in the stack.

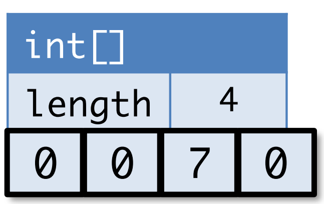
Type	Description	Values
`byte`	8-bit	`-128 to 127`
`short`	16-bit integer	`-32768 to 32767`
`int`	32-bit integer	-2^{31} to $2^{31} - 1$
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit IEEE floating point	
`double`	64-bit IEEE floating point	
`boolean`	true or false	`true false`
`char`	16-bit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

Reference Values stored on the Heap

Arrays

- Type of the array
- Length
- Values for all array **elements**

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

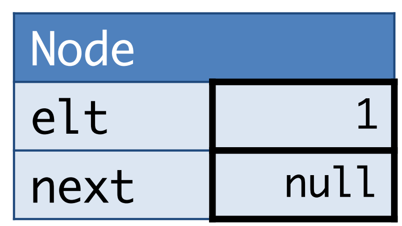


length *never* mutable;
elements *always* mutable

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?

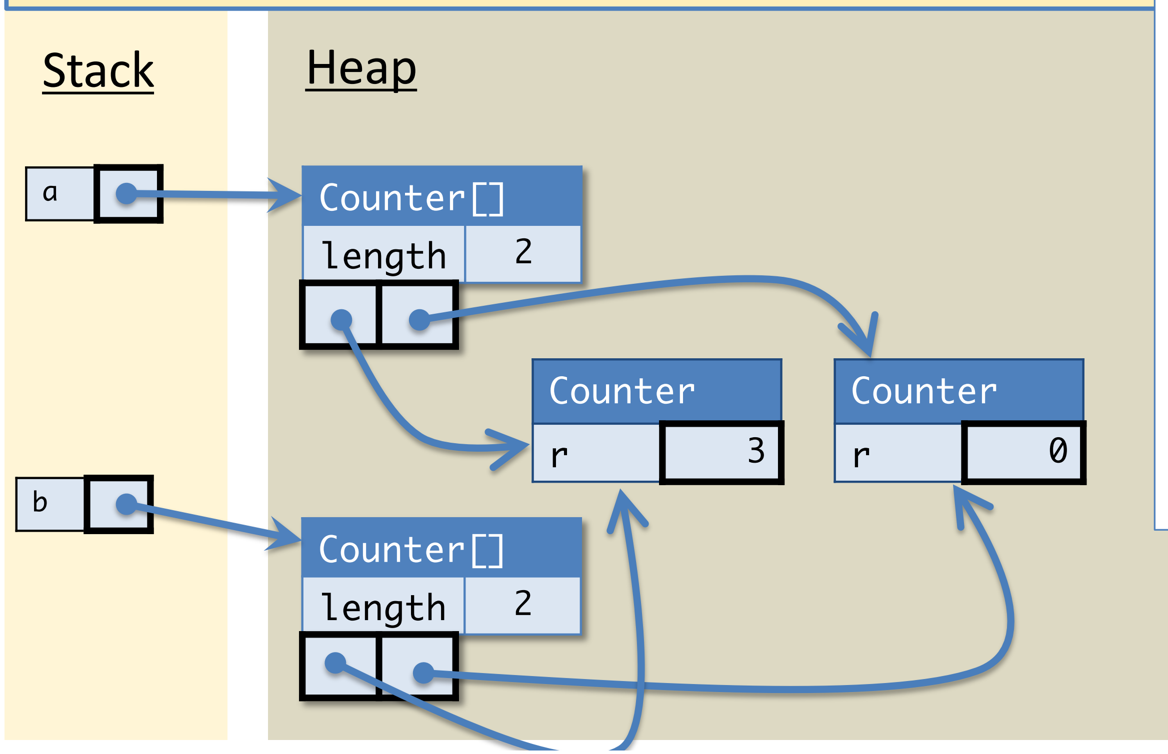
```
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

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

}
```

What does the 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;
  }

}
```



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

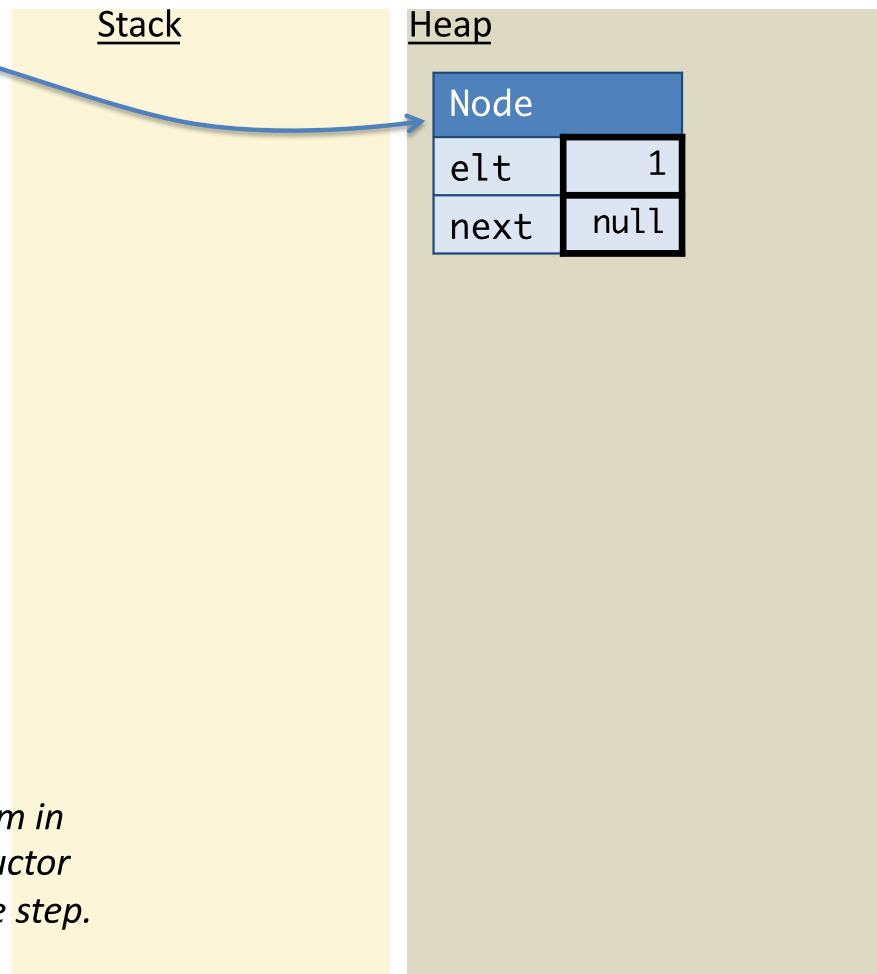
```
Node n1 = new Node(1,null);
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Stack

Heap

Workspace

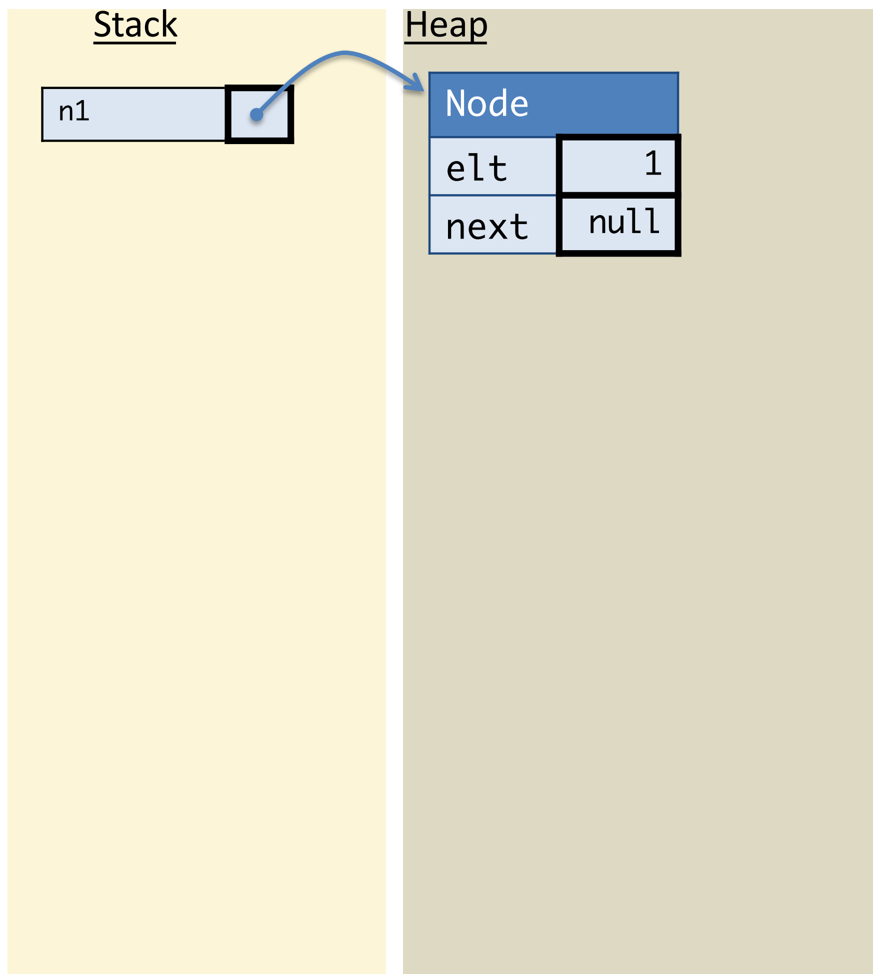
```
Node n1 = ;
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

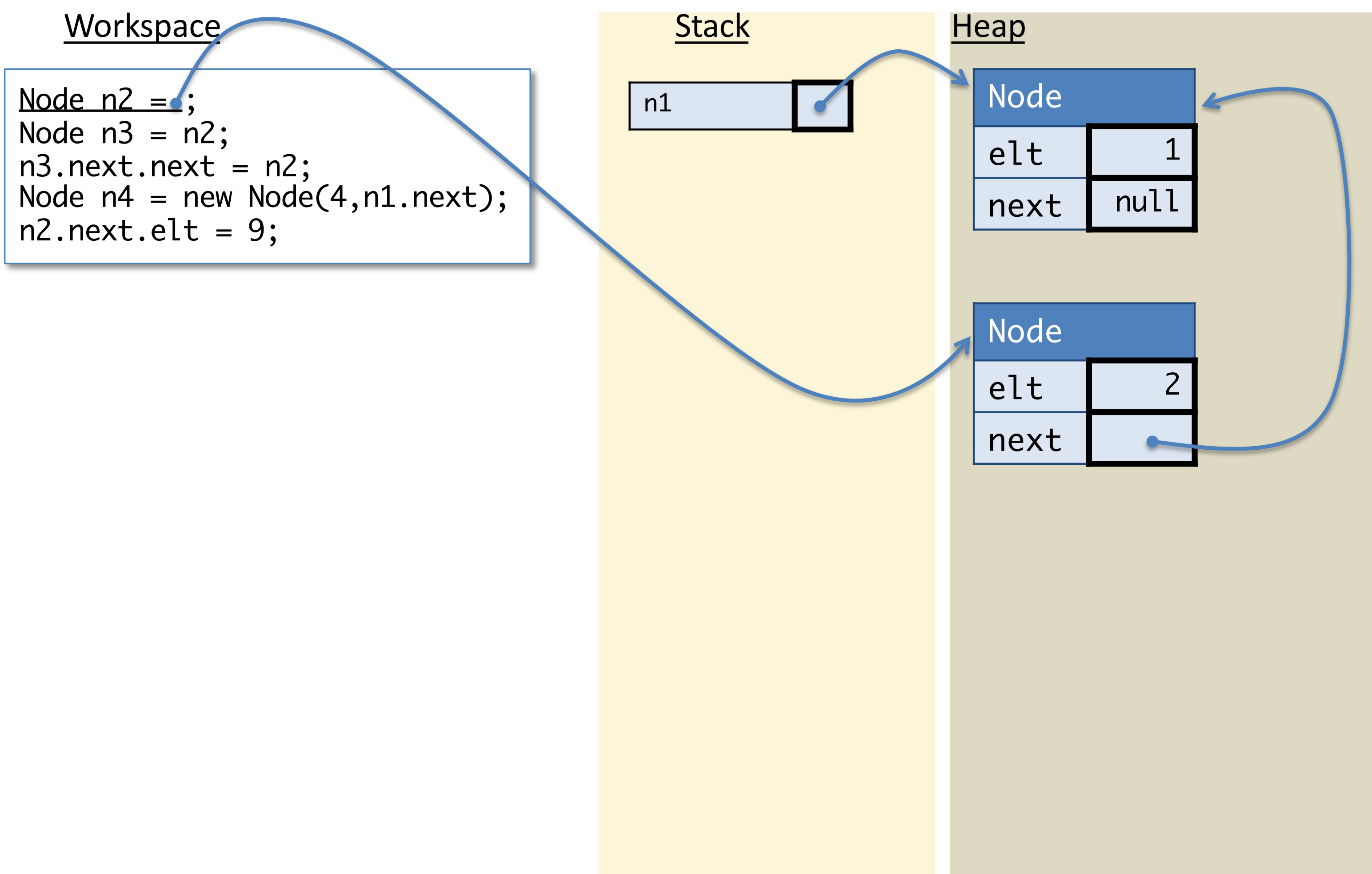


Note: we're skipping details here about how the constructor works. We'll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.

Workspace

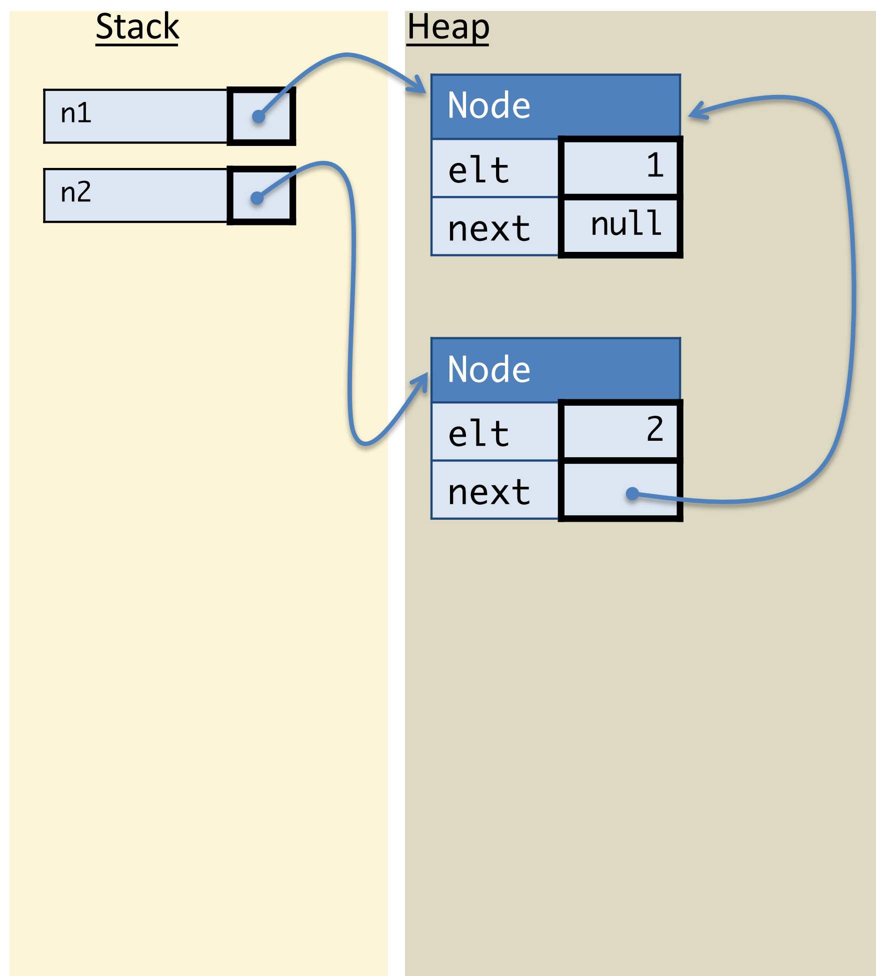
```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```




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

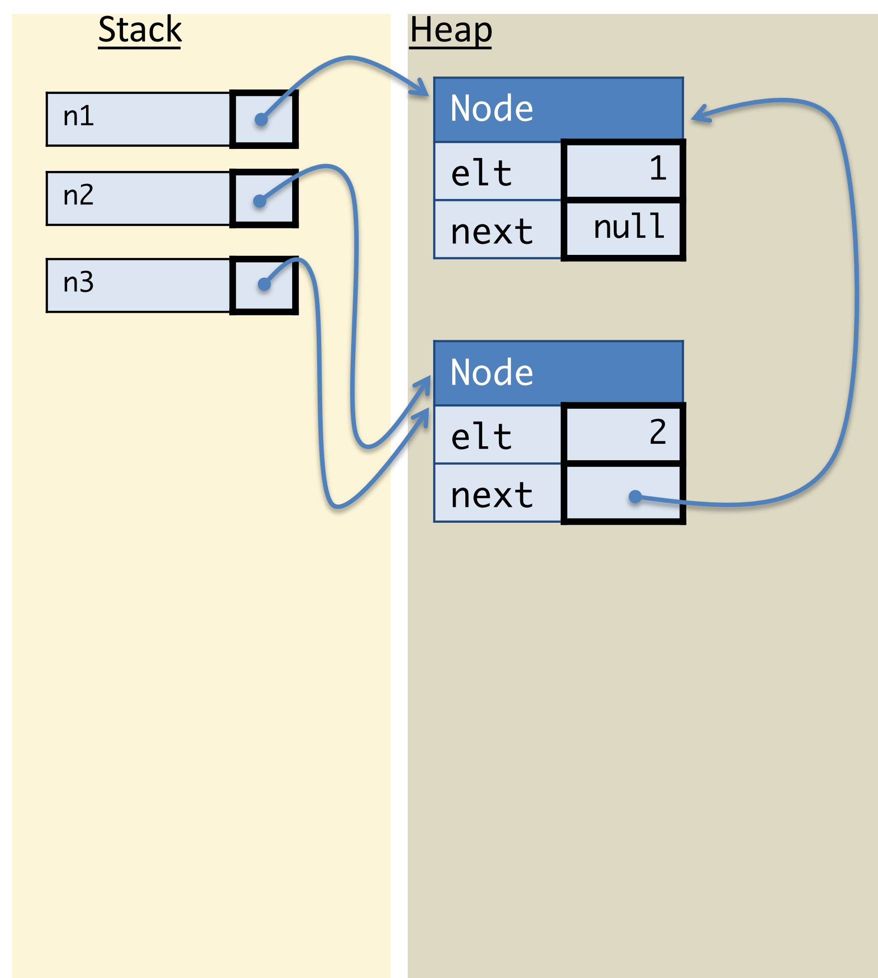
Workspace

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



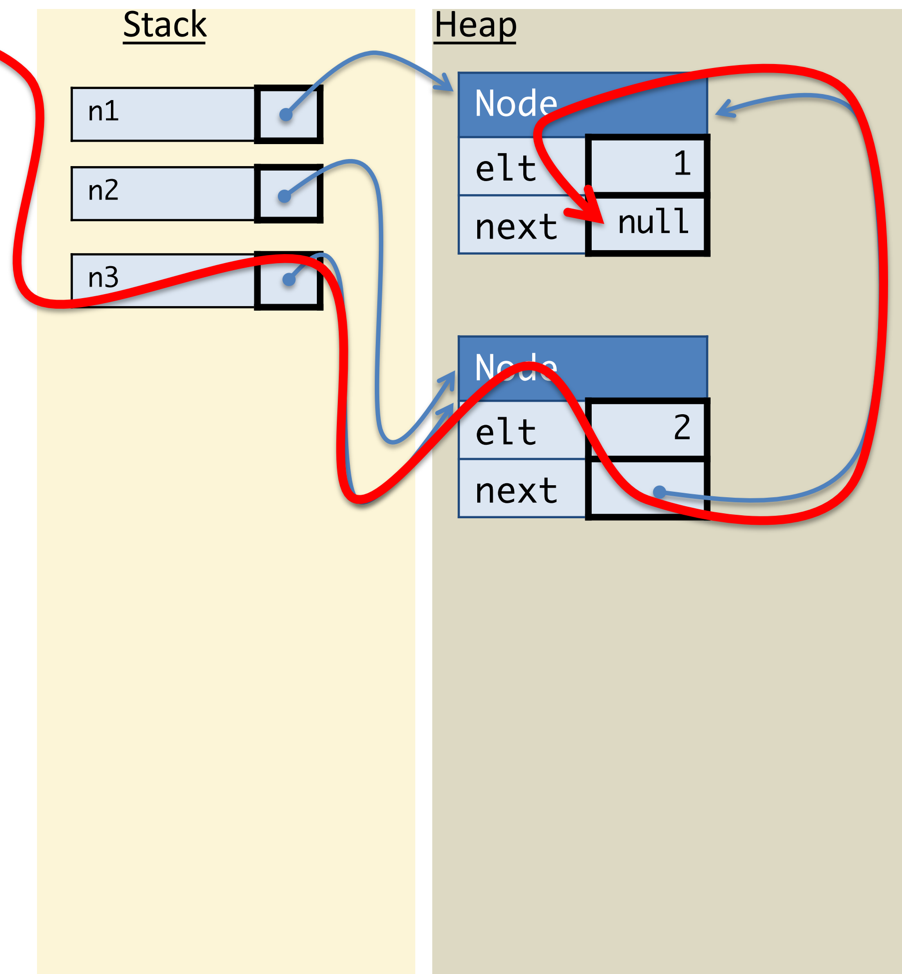
Workspace

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



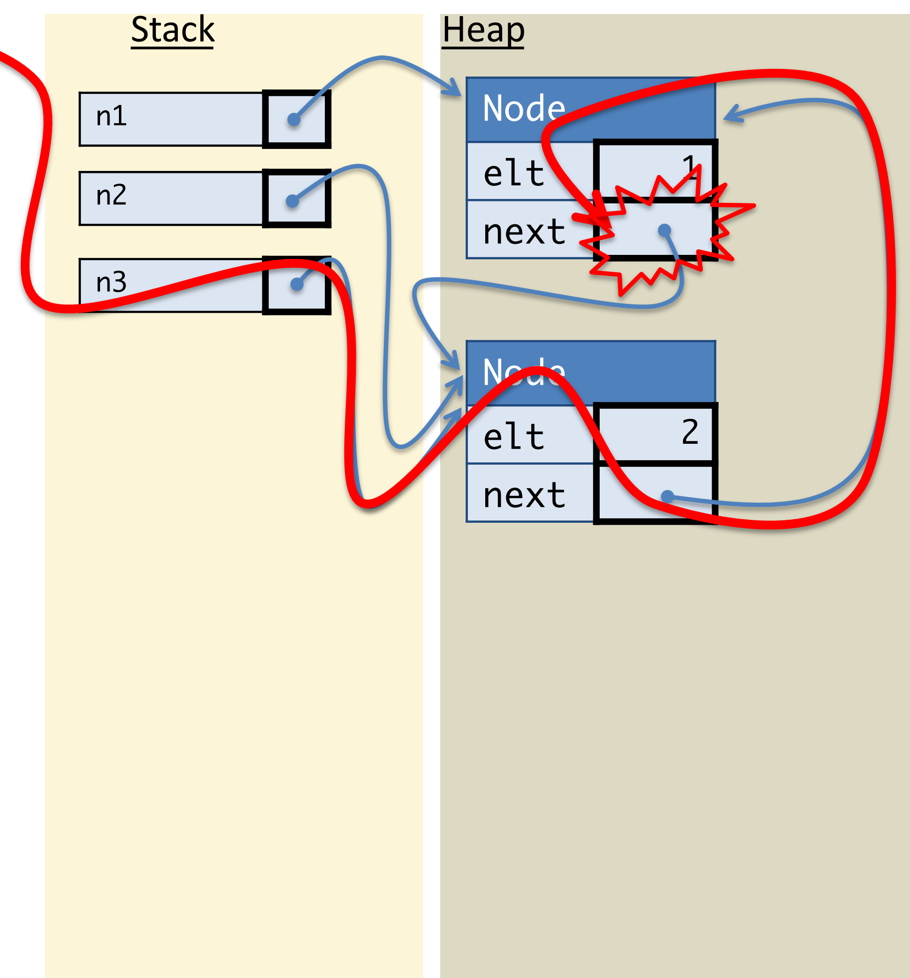
Workspace

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



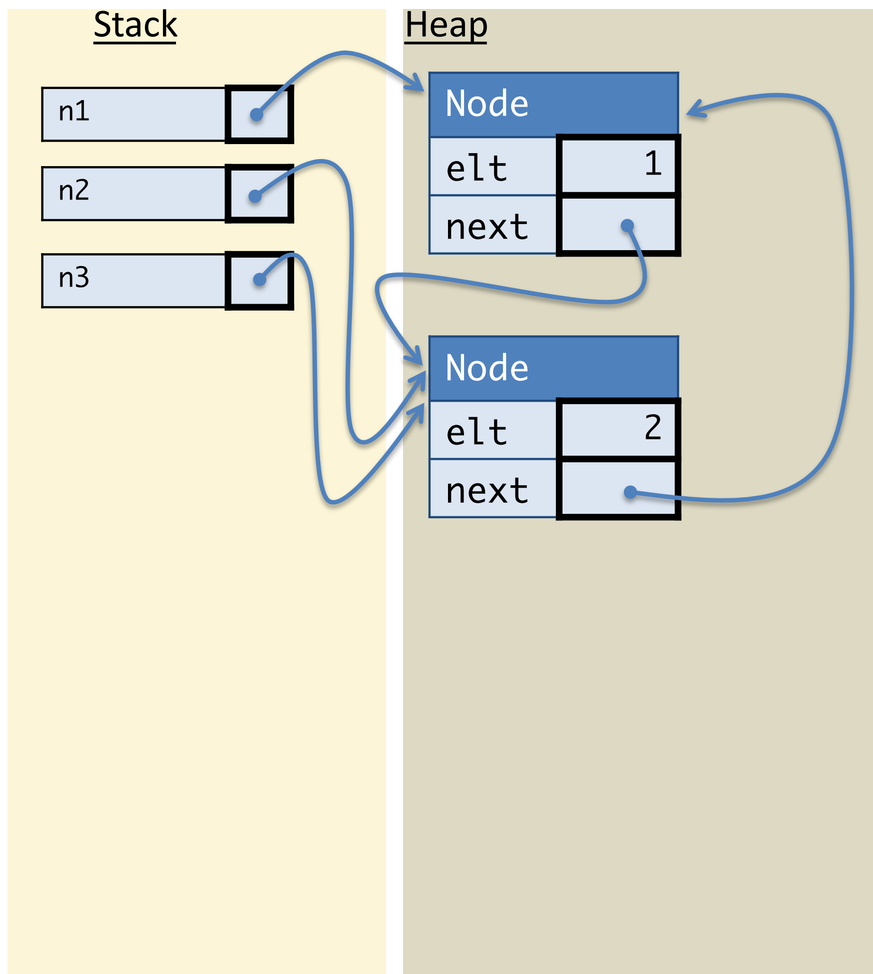
Workspace

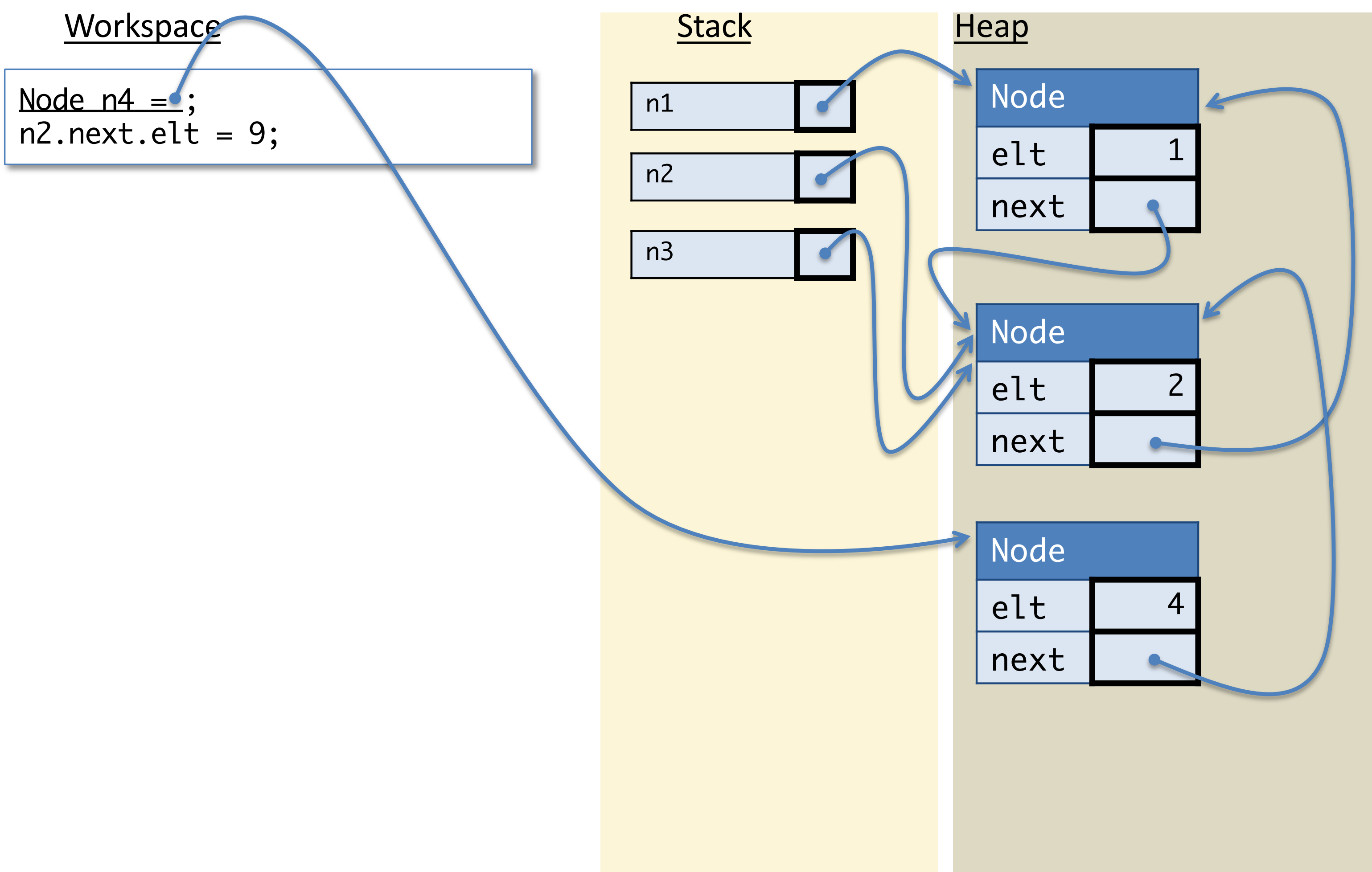
```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```



Workspace

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

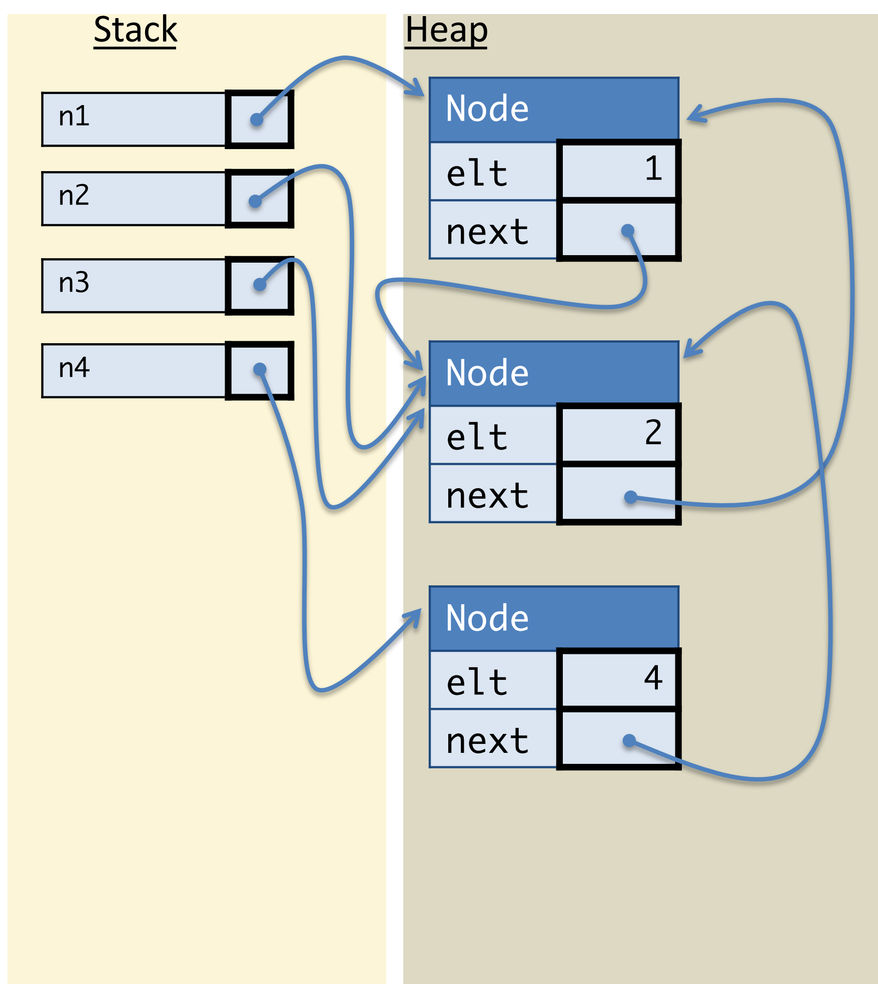


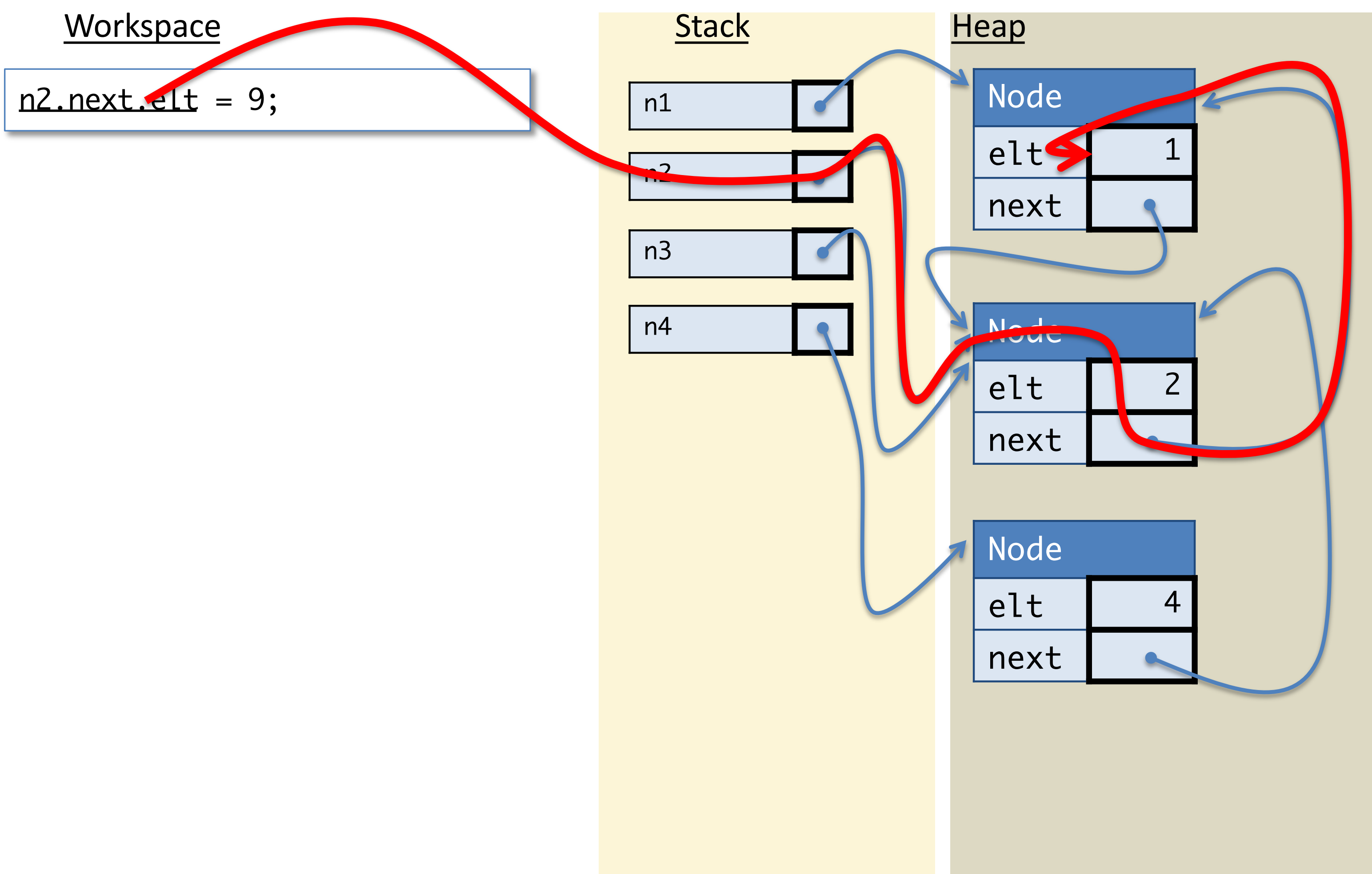


Workspace
Node n4 = ;
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

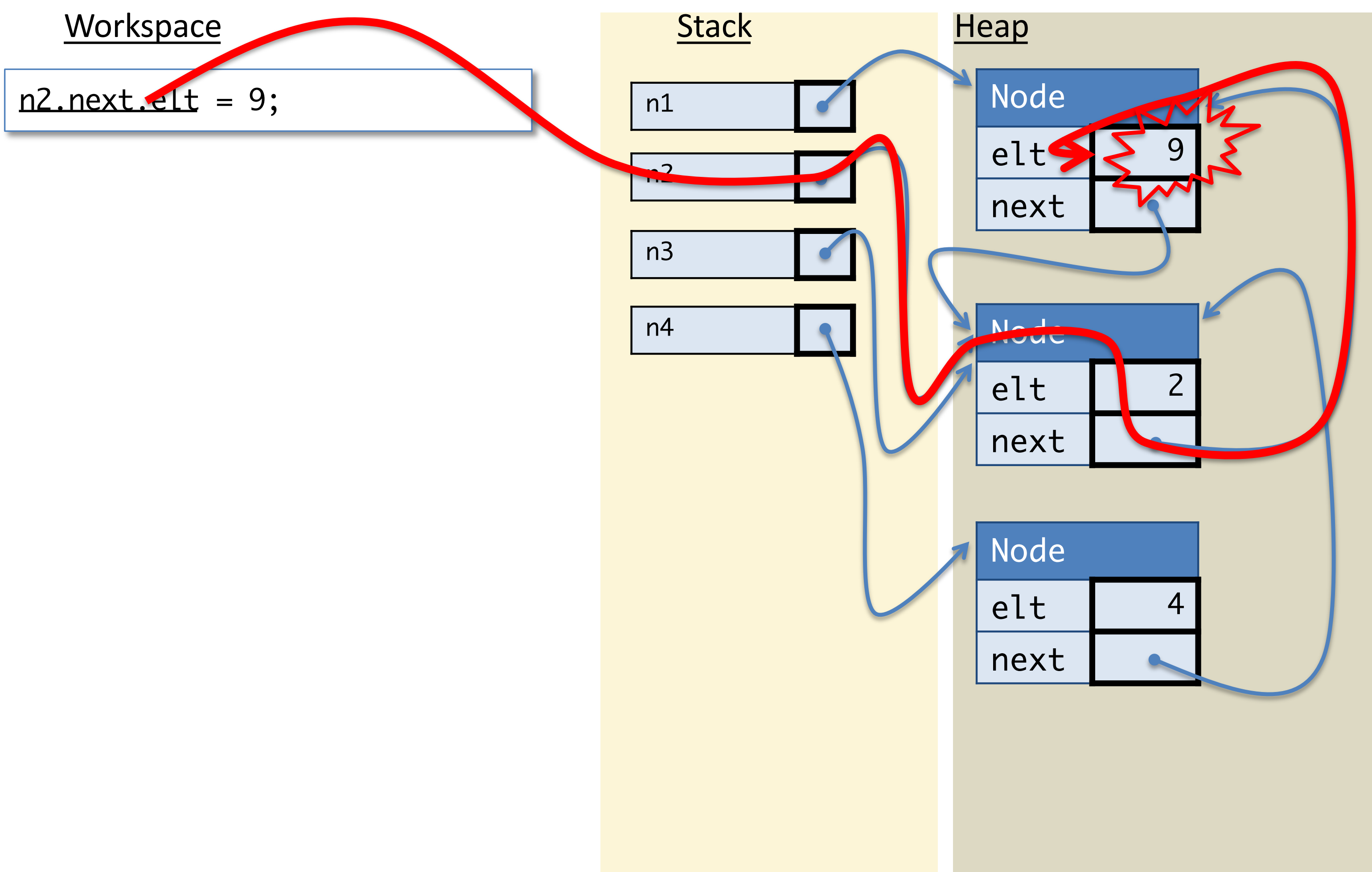
Workspace

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

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 to mutable private data from methods.

Encapsulation

```
public class C {
    private int x = 3;
    private int[] y = { 1, 2, 3 };
    public int getX() { return x; }
    public int[] getY() { return y; }
}
```

The instance variable x is **encapsulated** --- it can *only* be modified by the class C.
The instance variable y is **not encapsulated**. Code in *any class* can modify the values stored in the array.

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`
  - `new Counter()` has type `Counter`
- How do we know if `x.inc()` 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, objects 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

```
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

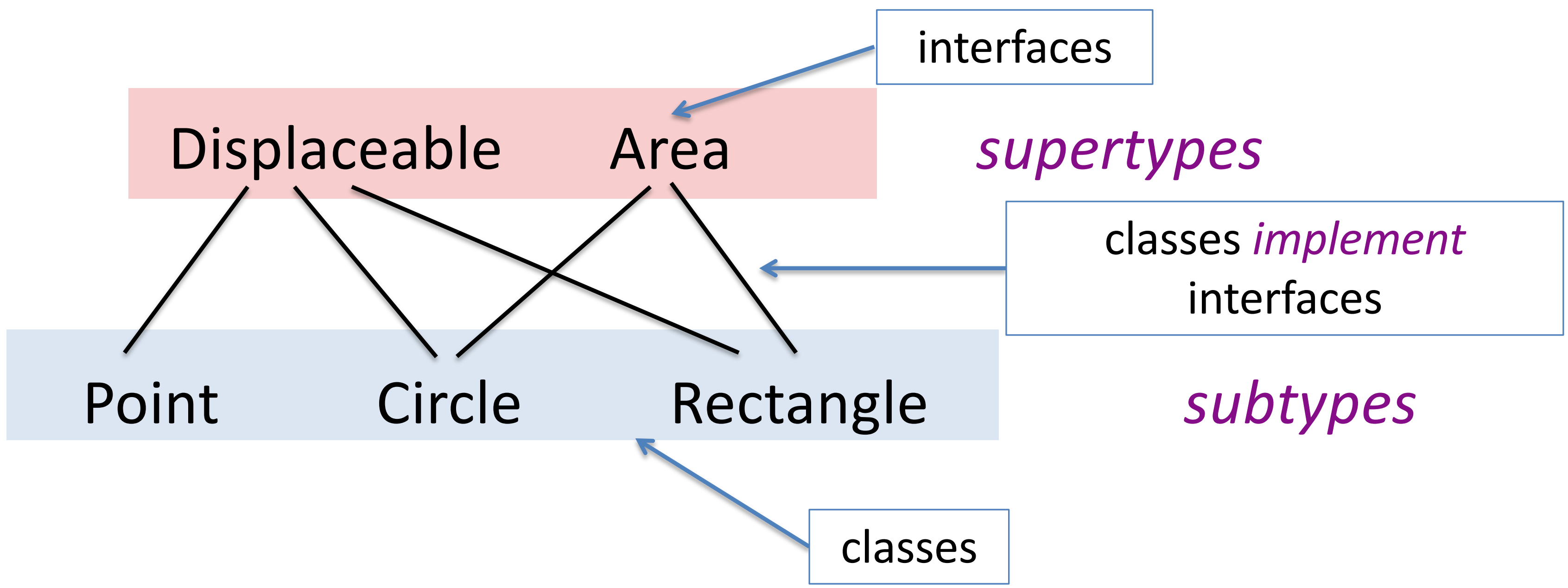
Subtyping

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

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*

- Key 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
- Potential confusion: subtypes have **more methods** than supertypes. (There are **more objects** that belong to the supertype than the subtype.)

**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

Key Idea: Liskov's *Substitution Principle**

If S is a subtype of T, then an object of type T may be replaced by an object of type S anywhere a T is expected.

- **without** changing the properties of the program

*Named for Turing award winner and designer of the influential OO language CLU, Barbara Liskov, who introduced this idea in 1988.

Extension – More complex subtyping

Extension – More complex subtyping

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

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

Interface Extension

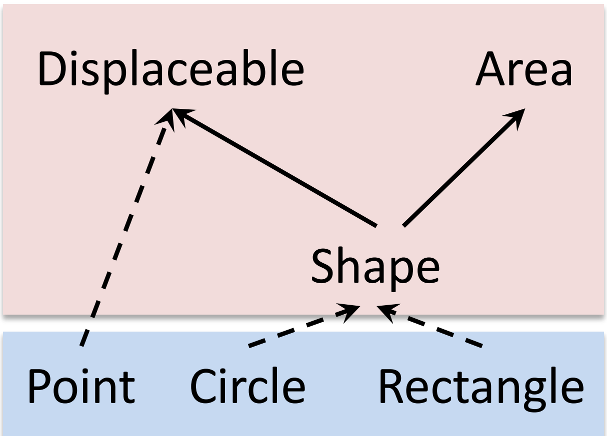
- Build richer interface hierarchies by *extending* existing interfaces.

```
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; both are also subtypes of Displaceable and Area *by transitivity*.
- Note that one interface may extend *several* others.
  - Interfaces do not necessarily form a tree, but the interface hierarchy cannot have any cycles.

Class Extension: “Inheritance”

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

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

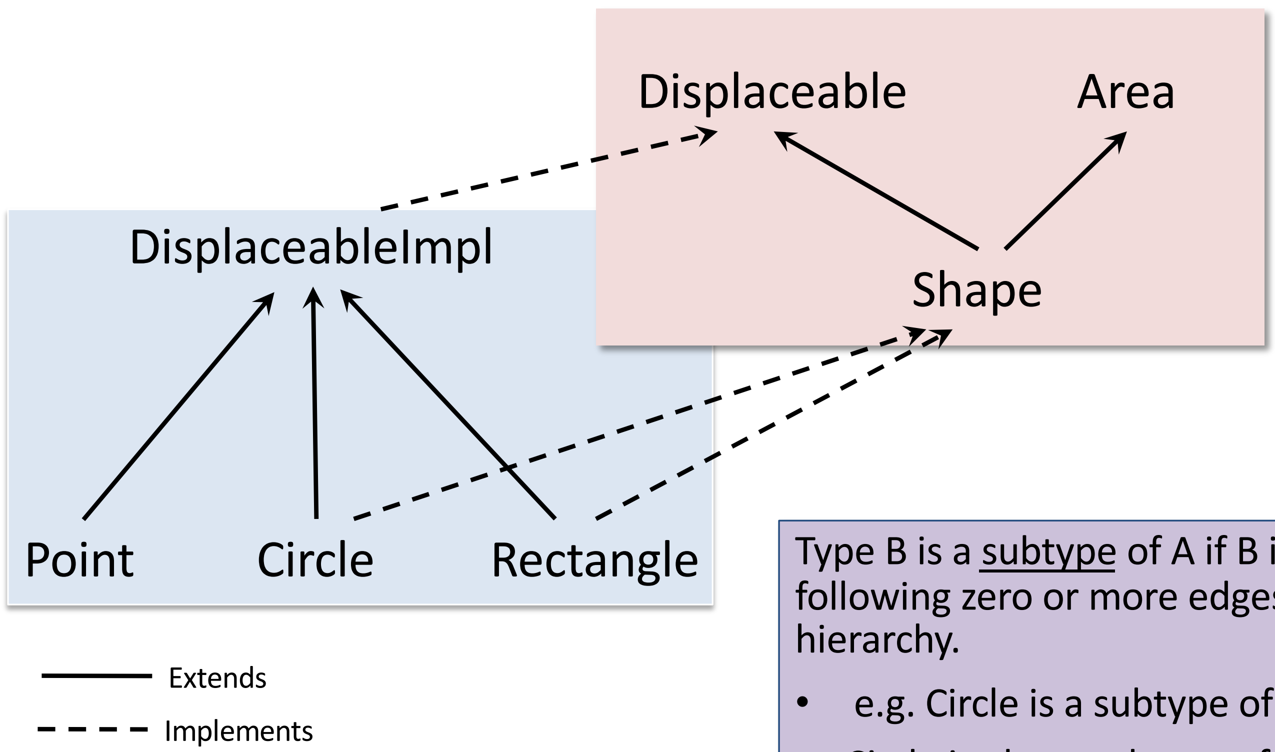
Simple Inheritance

- In *simple inheritance*, the subclass only *adds* new fields or methods.
  - It is also possible to replace (override) method definitions – we'll come back to this later
- 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 B is a <u>subtype</u> of A if B is reachable from A by following zero or more edges upwards in the hierarchy.

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