

# Programming Languages and Techniques (CIS1200)

## Lecture 25

Java ASM, Subtyping and Extension  
Chapters 23 & 24

# Announcements

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

# Revenge of the Abstract Stack Machine

# 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 *what class was used to create them*
  - *Arrays store type information*
  - *New ASM component: Class table (coming soon)*

# Java Primitive Values

The values of these data types occupy one machine word (or less) 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'

# 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 non-static fields

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

Node	
elt	1
next	null

fields may  
or may not be  
Mutable;  
public/private  
annotations 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;  
    }  
  
}
```



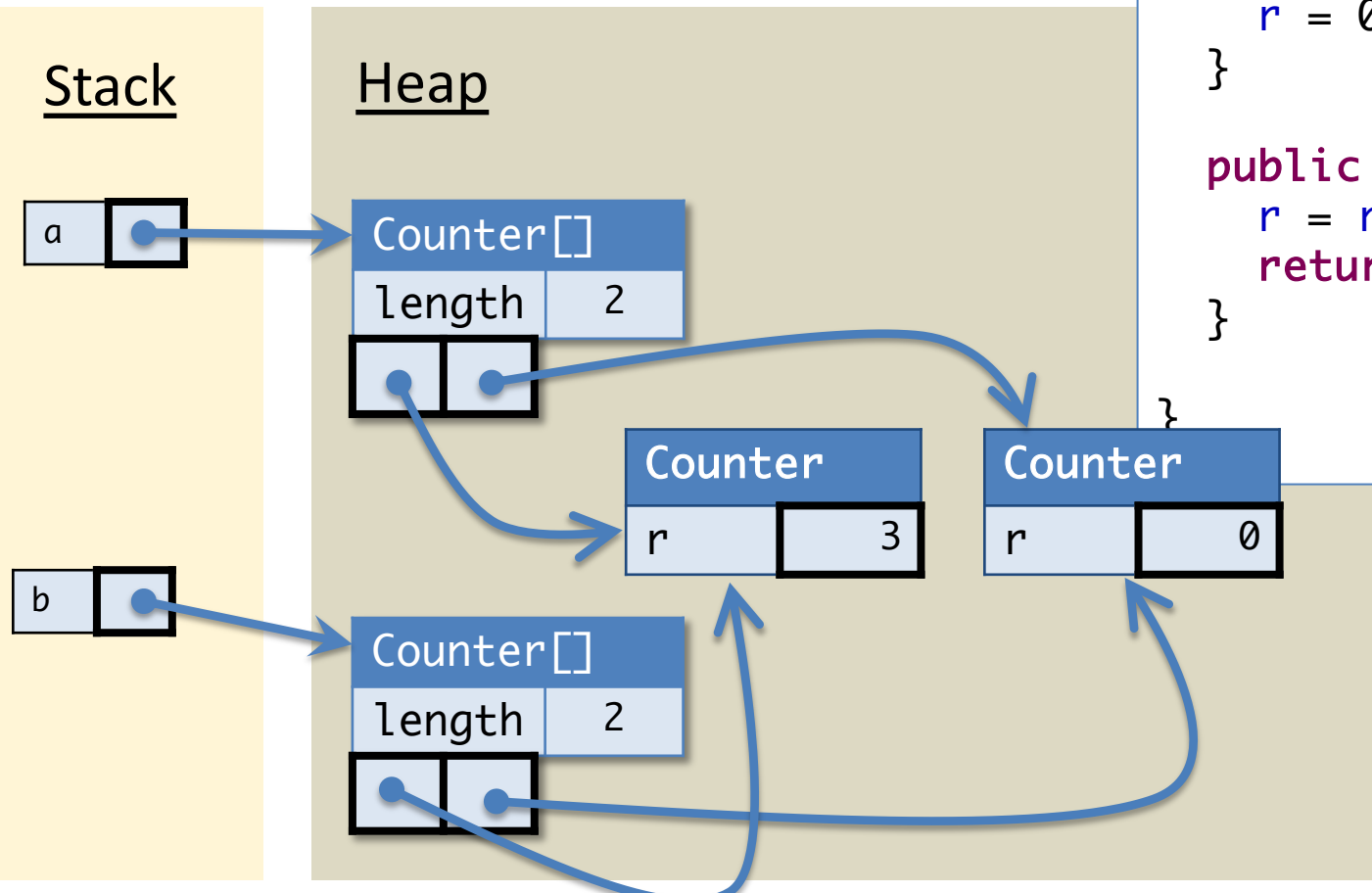
What does the ASM look like at the end of this program?

```
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;  
    }  
}
```

Stack

Heap



## 24: 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 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.elc = 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.elc = 9;
```

## Stack

## Heap

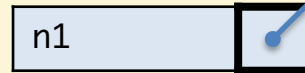
Node	
elt	1
next	null

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

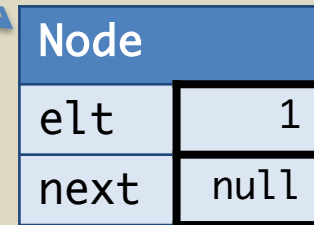
## Workspace

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

## Stack



## Heap



## Workspace

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

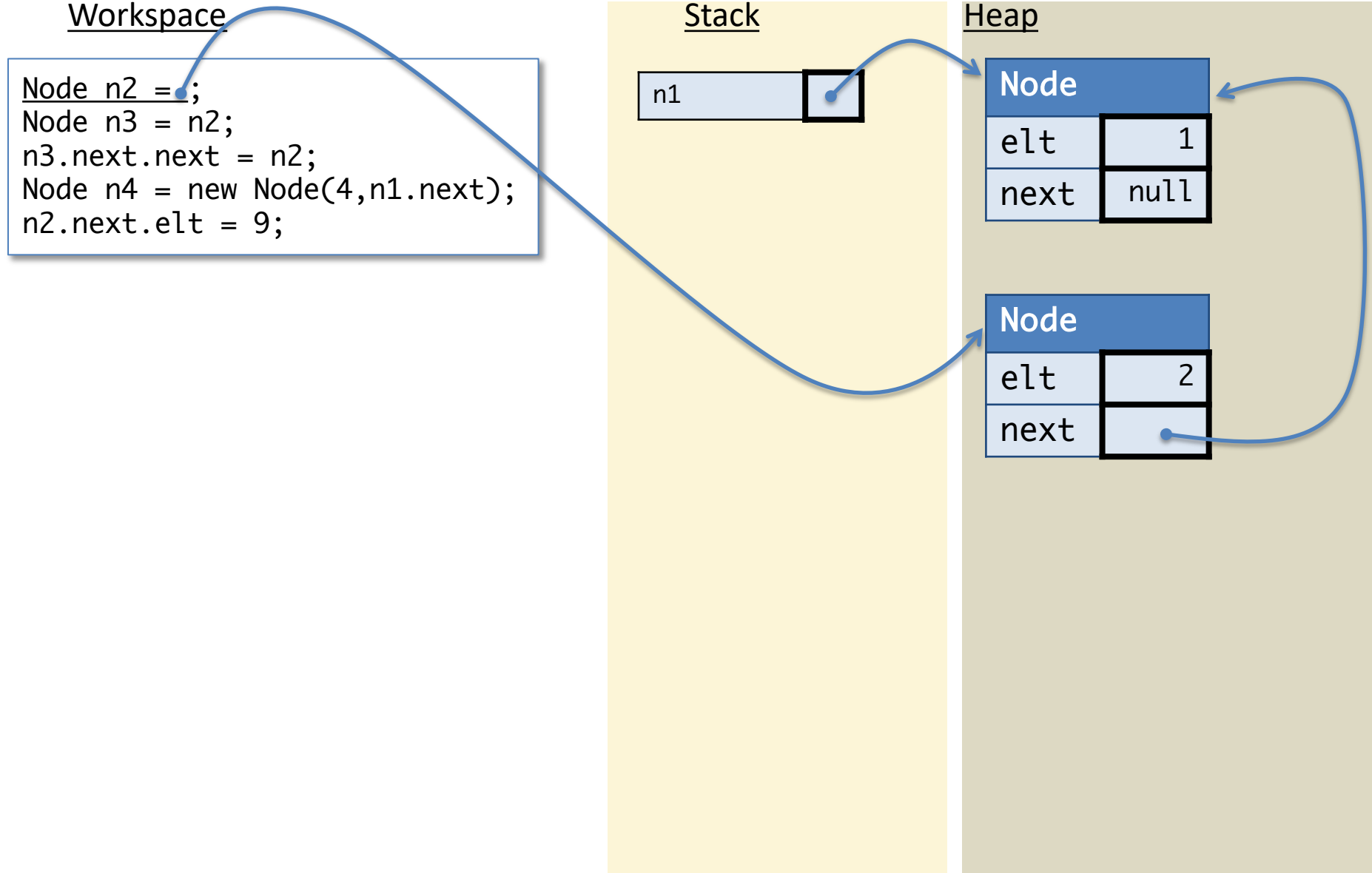
## Stack

n1

## Heap

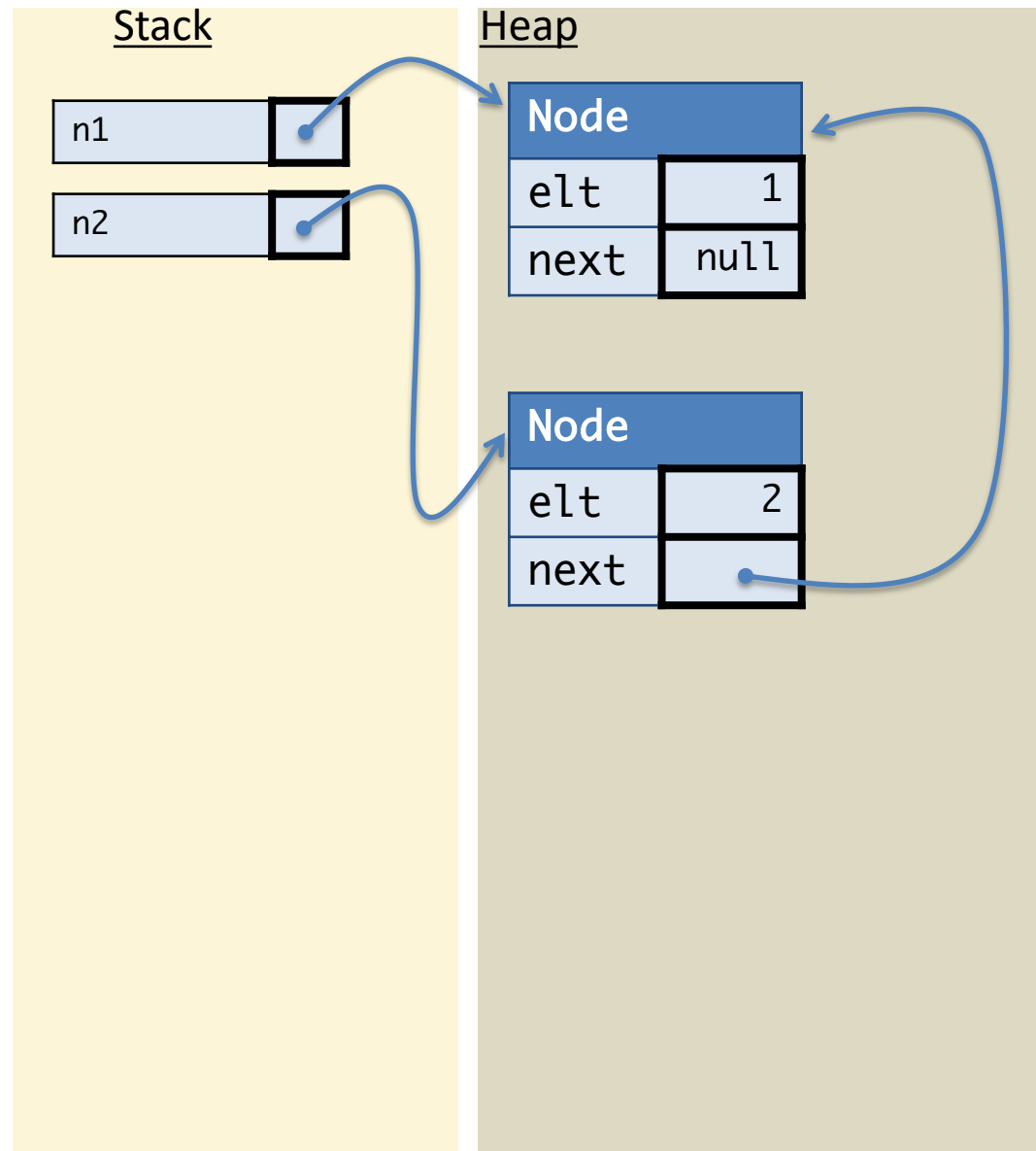
Node	
elt	1
next	null

Node	
elt	2
next	



## Workspace

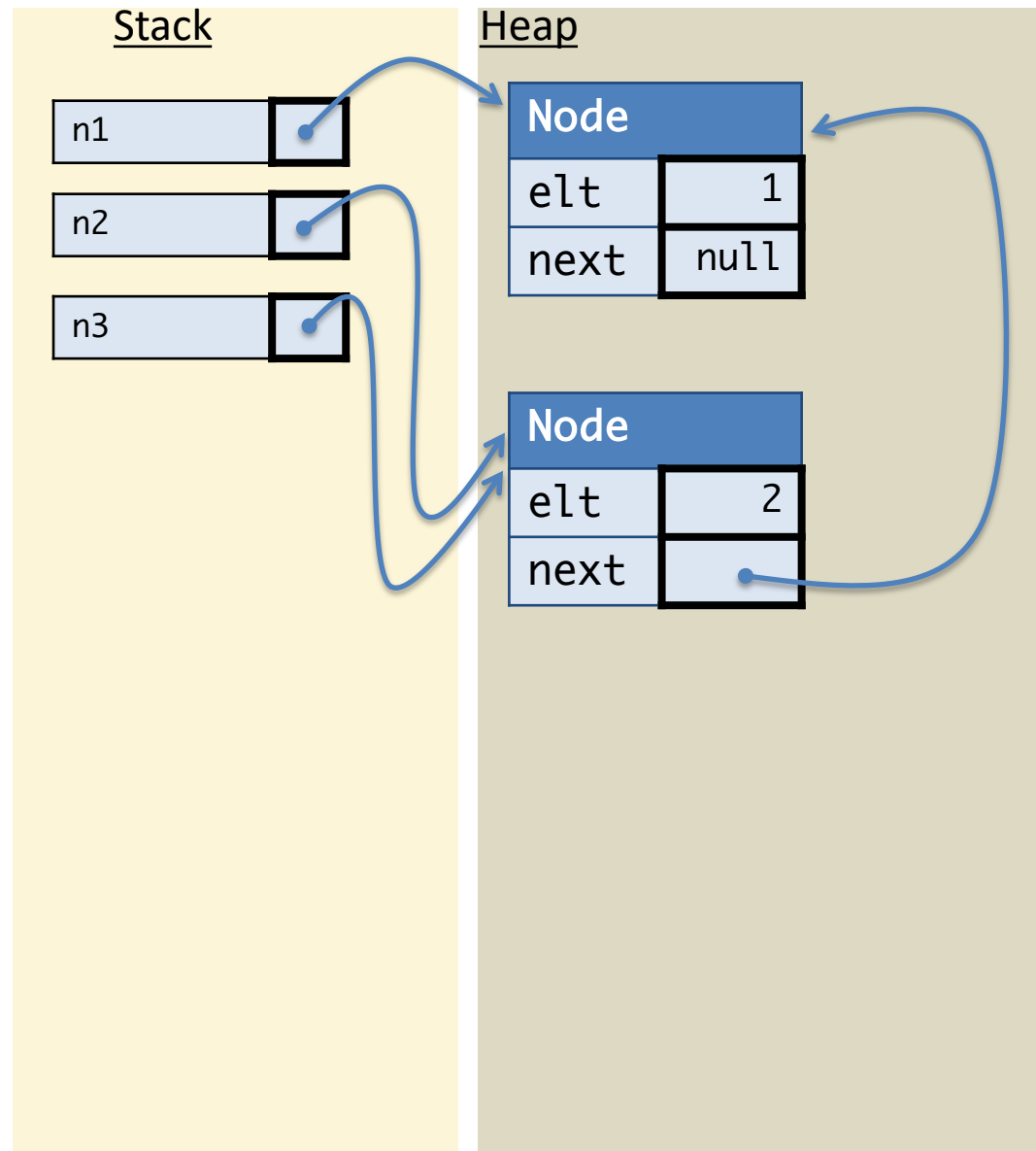
```
Node n3 = n2;  
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next.elc = 9;
```





## Workspace

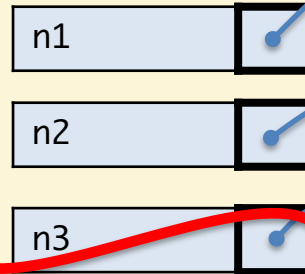
```
n3.next.next = n2;  
Node n4 = new Node(4,n1.next);  
n2.next.elc = 9;
```



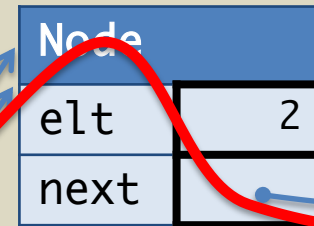
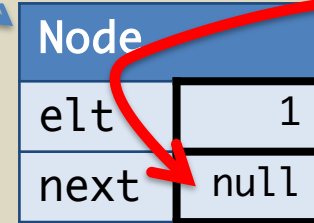
## Workspace

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

## Stack



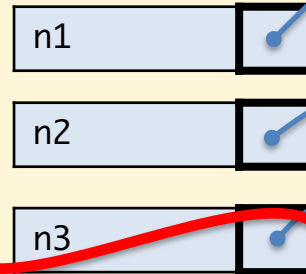
## Heap



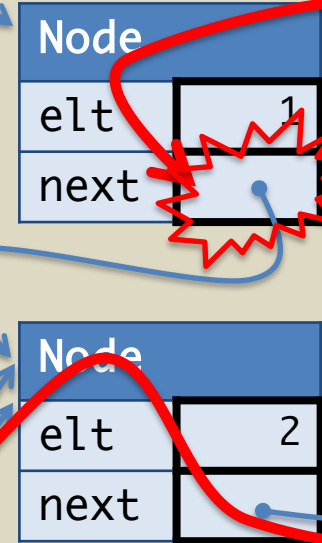
## Workspace

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

## Stack

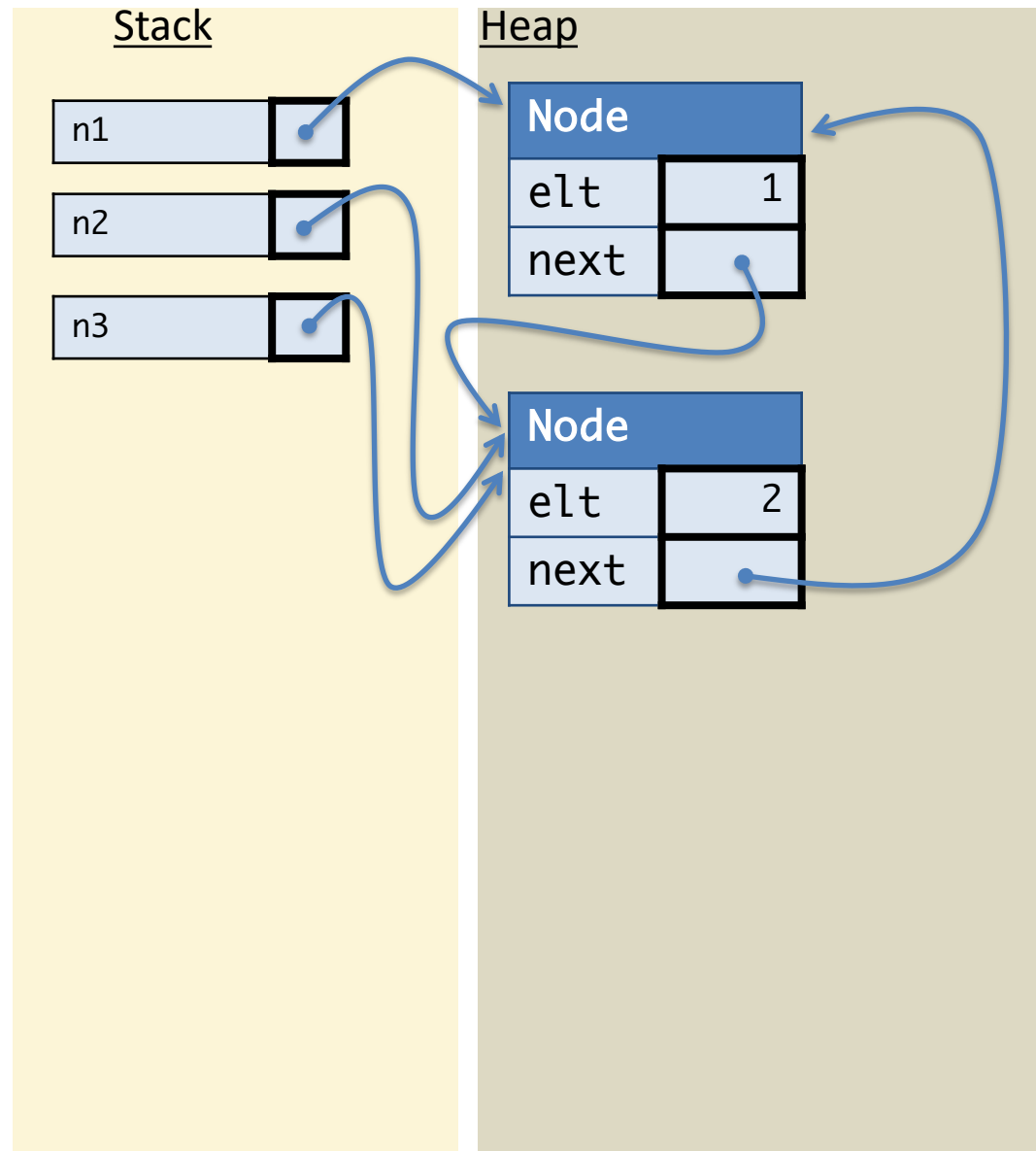


## Heap



## Workspace

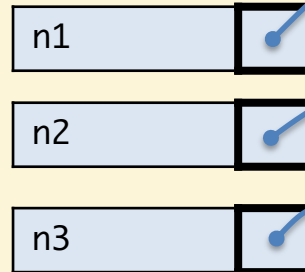
```
Node n4 = new Node(4,n1.next);  
n2.next.elc = 9;
```



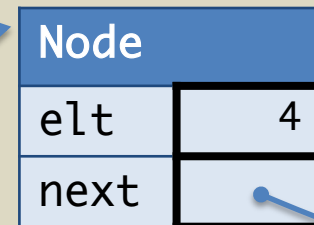
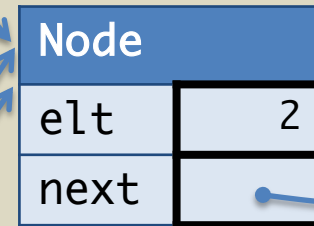
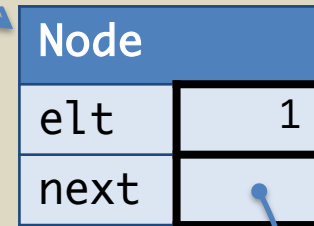
## Workspace

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

## Stack

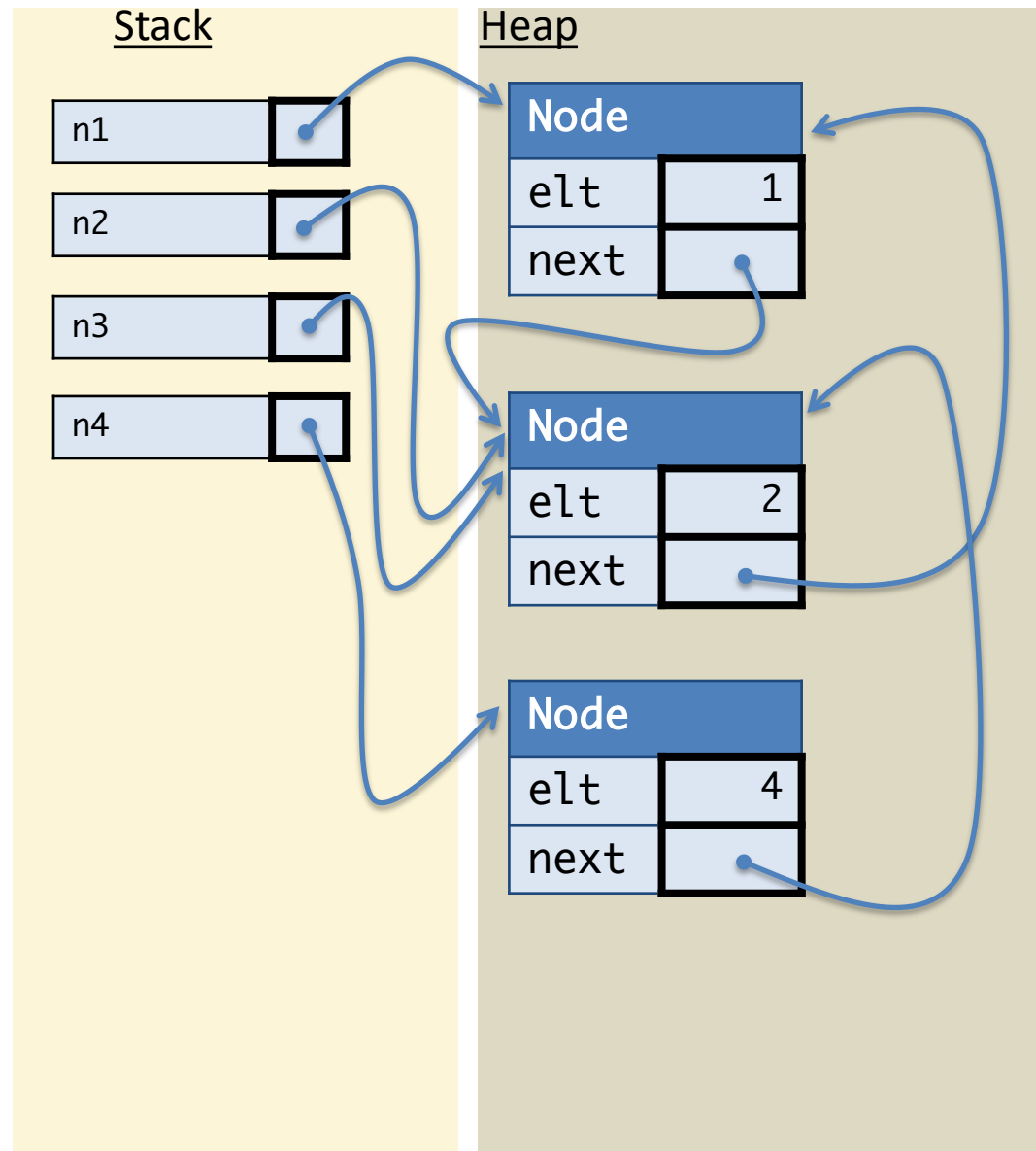


## Heap



## Workspace

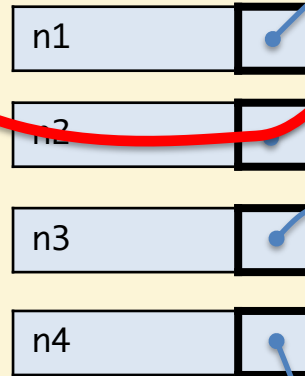
```
n2.next.elc = 9;
```



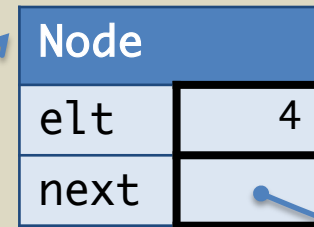
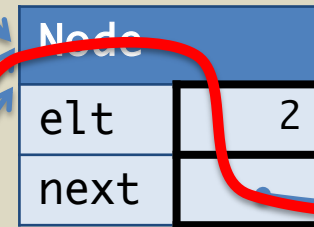
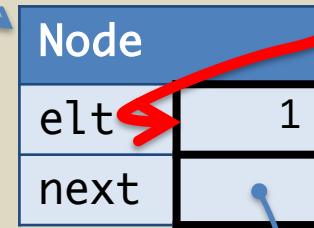
## Workspace

```
n2.next.elc = 9;
```

## Stack



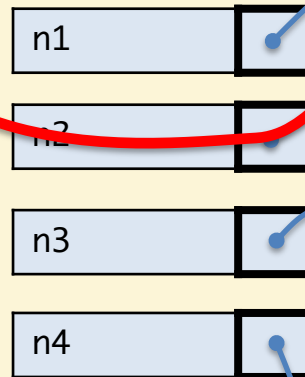
## Heap



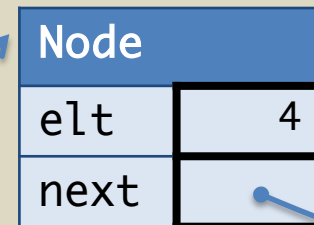
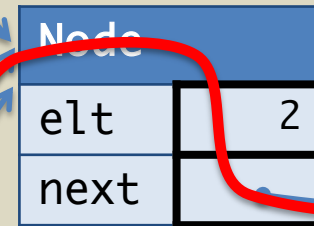
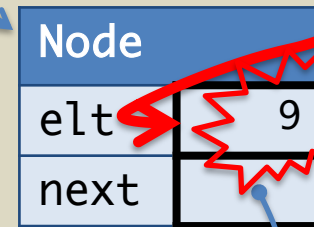
## Workspace

```
n2.next.elc = 9;
```

## Stack



## Heap

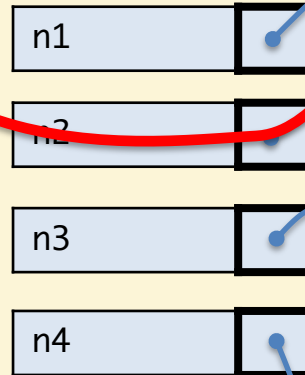




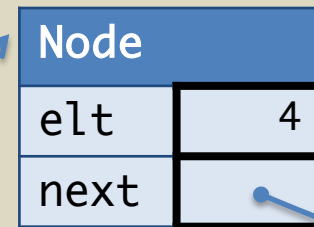
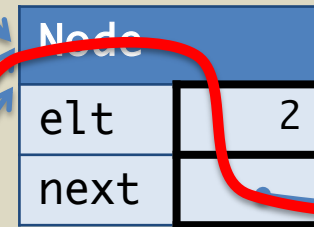
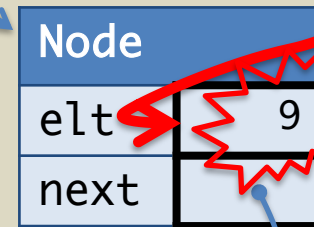
## Workspace

```
n2.next.elc = 9;
```

## Stack



## Heap

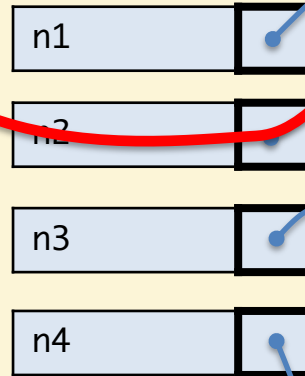


*So if we now print the value of `n1.elc`, we get...*

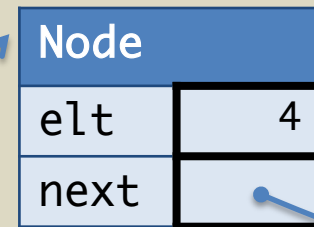
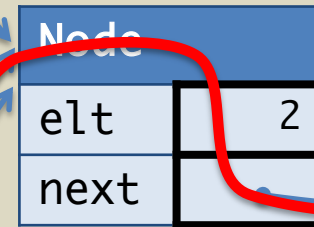
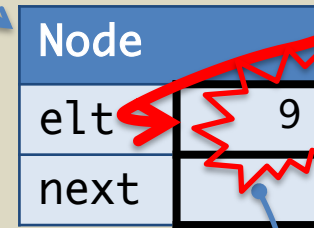
## Workspace

```
n2.next.elc = 9;
```

## Stack



## Heap



*So if we now print the value of `n1.elc`, we get...*

9

Whew.

OOoooooo programming



OO

Subtypes

# 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
  - Assignments like `"x = 3"` must be of values with compatible types
  - methods `"o.m(3)"` must be called with compatible arguments

HOWEVER: in Java, values can have *multiple* types....

# Review: 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);  
}
```

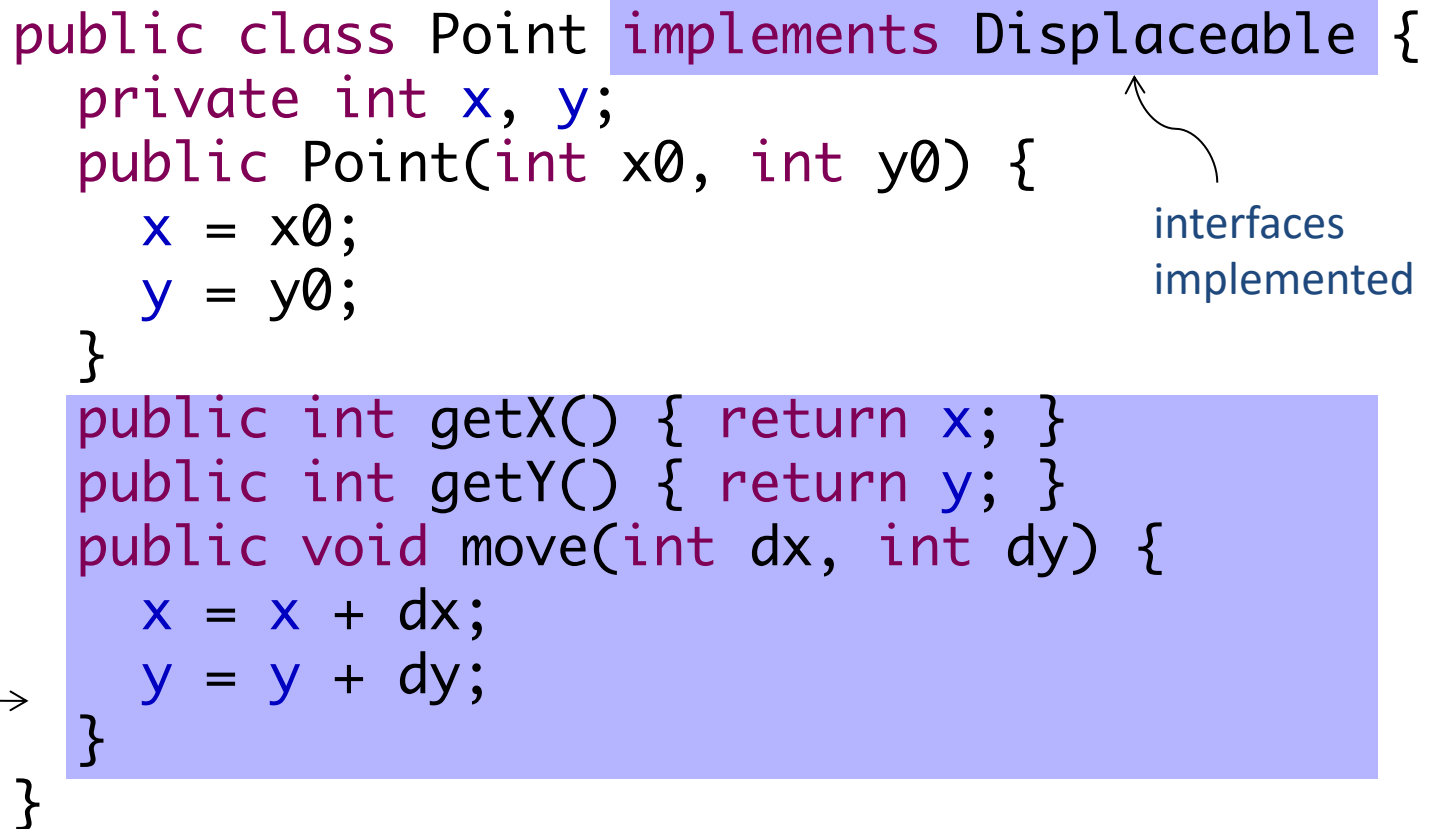


No fields, no constructors, no  
method bodies!

# Implementing an 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) {  
        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:



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

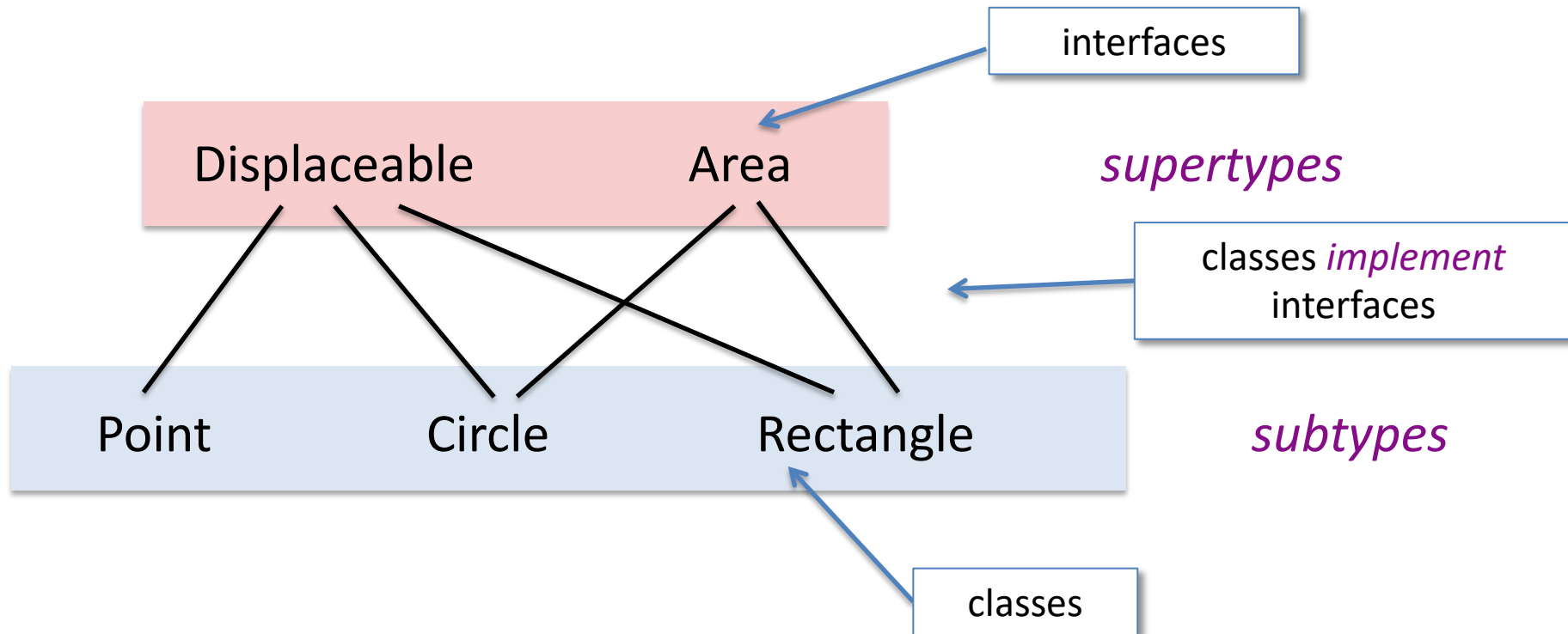
**Definition:** Type A can be declared to be a *subtype* of type B if values of type A can do anything that values of type B can do. Type B is called a *supertype* of A.

**Example:** A class that implements an interface

Subtyping relationships are **explicitly declared** in Java

# 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 non-static 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

## 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 breaking the program's type-correctness)



\*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);  
}
```

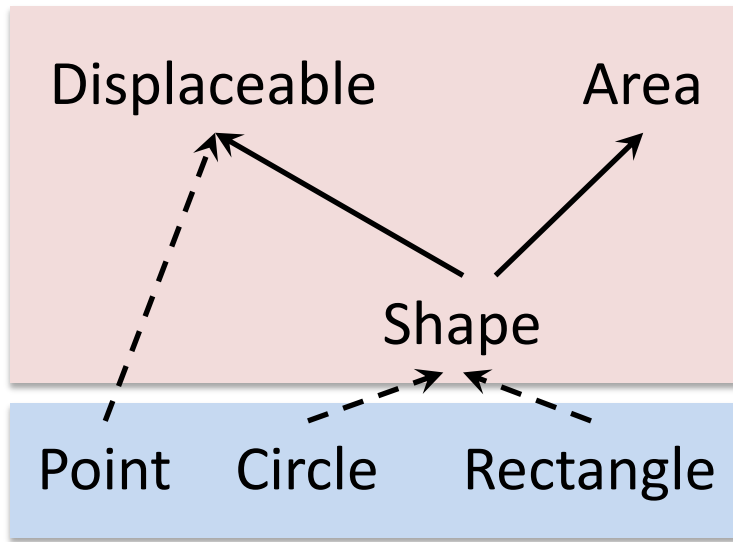
```
public interface Area {  
    double getArea();  
}
```

```
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

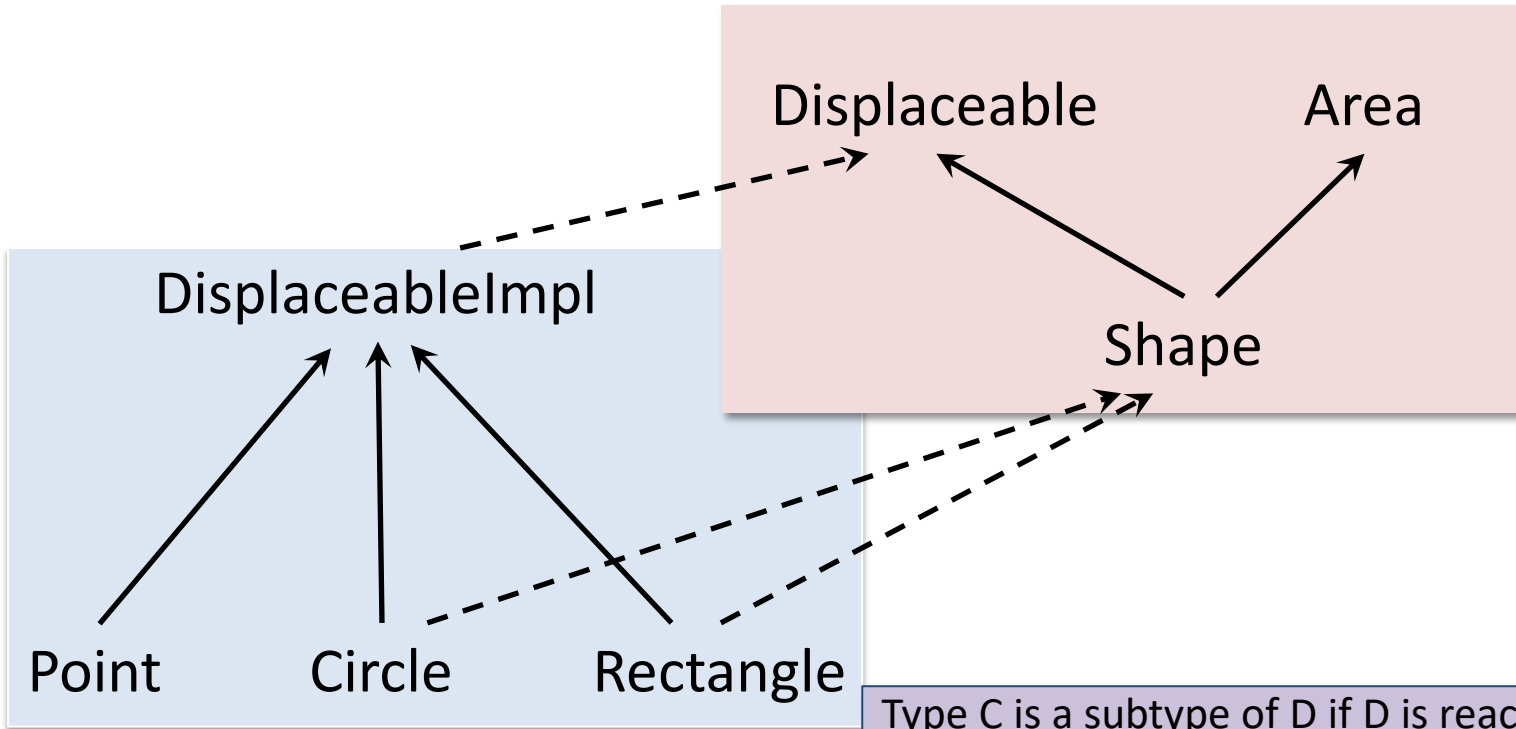
- In *simple inheritance*, the subclass only *adds* new fields or methods.
  - It is also possible to replace (override) method definitions – we'll come back to this later
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for 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



—— Extends  
- - - Implements

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
- Circle is also a subtype of *itself*



# Example of Simple Inheritance

See: [Shapes.zip](#)

# Inheritance: Constructors

- Constructors 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)

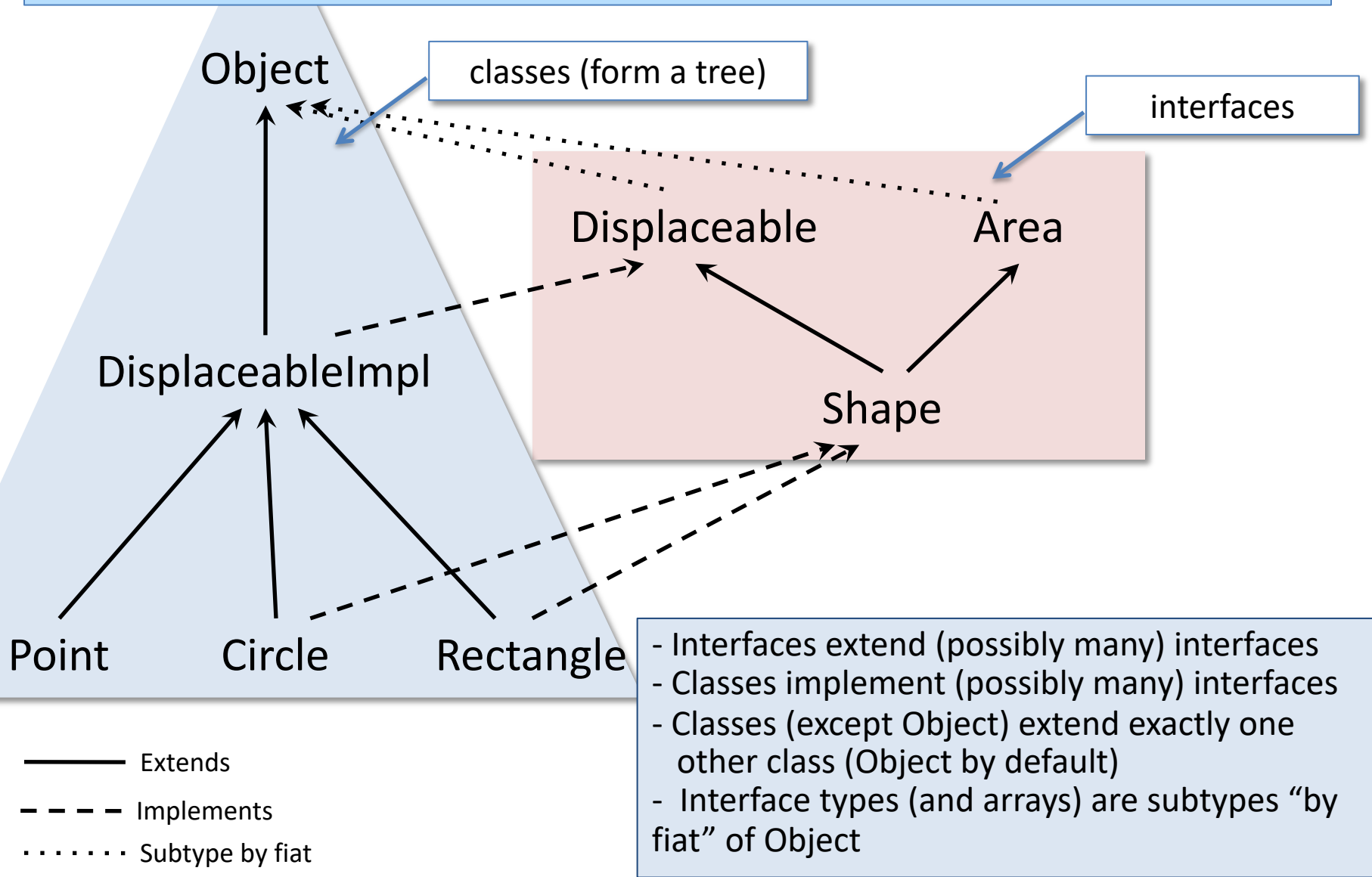
```
public Circle(Point pt, int radius) {  
    super(pt.getX(),pt.getY());  
    this.radius = radius;  
}
```

# 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
  - The Object class provides methods useful for *all* objects to support
- Object is the top (i.e., “most super”) type in the subtyping hierarchy

# Recap



# 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 using it is part of a well-known "contract" of the design*

# Static Types vs. Dynamic Classes

# "Static" types vs. "Dynamic" classes

- The *static type* of an *expression* is a type that describes what we 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 created from at run time

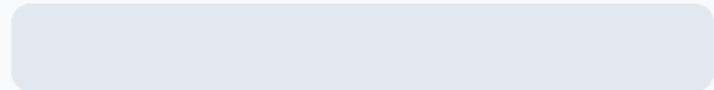
x = new Point(2,3)

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

25: What is the static type of  $a1$  on line A?

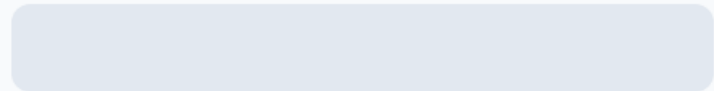


Area



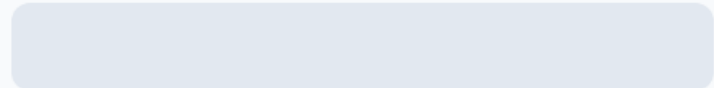
0%

Circle



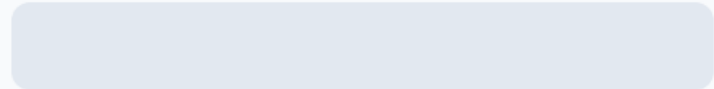
0%

None of the above



0%

Not well typed



0%



# Static type vs. Dynamic type

```
public Area asArea (Area a)
{ return a; }
```

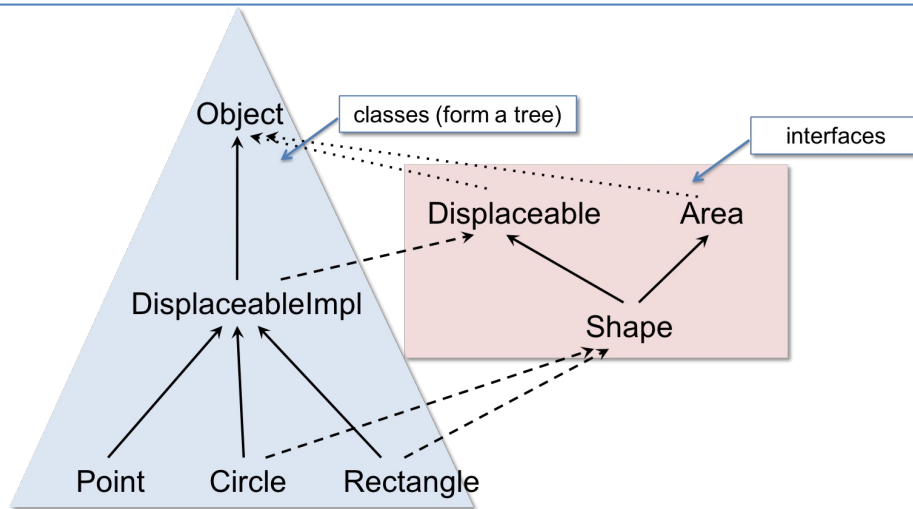
...

```
Point p = new Point(5,5);
Circle c = new Circle (p,3);
Area a1 = c; // A
```

```
--B-- y = asArea (c);
```

What is the static type of a1 on line A?

1. Area
2. Circle
3. None of the above
4. Not well typed

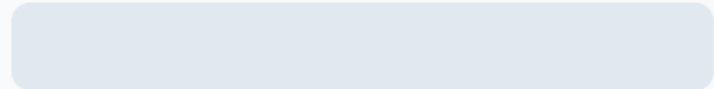


Area

25: What is the dynamic class of  $a1$  when execution reaches A?

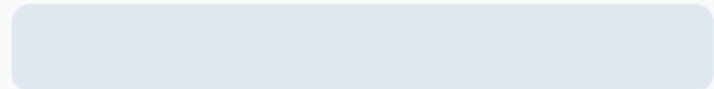


Area



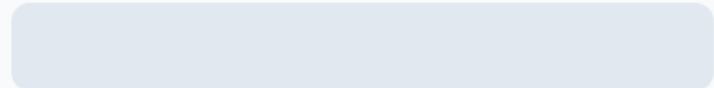
0%

Circle



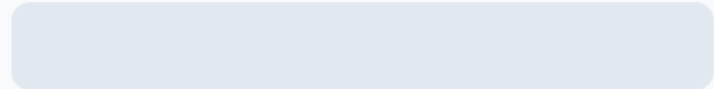
0%

None of the above



0%

Not well typed



0%

# Static type vs. Dynamic class

```
public Area asArea (Area a)
{ return a; }
```

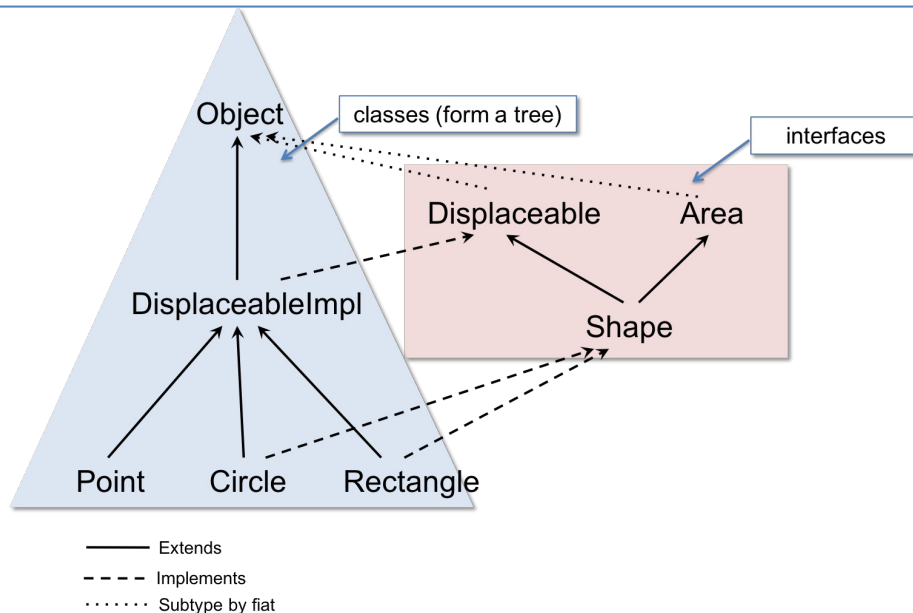
...

```
Point p = new Point(5,5);
Circle c = new Circle (p,3);
Area a1 = c; // A
```

```
--B-- y = asArea (c);
```

What is the dynamic class of a1 when execution reaches A?

1. Area
2. Circle
3. None of the above
4. Not well typed



Circle

25: What type could we declare for  $x$  (in blank B)?



Area

0%

Circle

0%

None of the above

0%

Not well typed

0%

# Static type vs. Dynamic class

```
public Area asArea (Area a)
{ return a; }
```

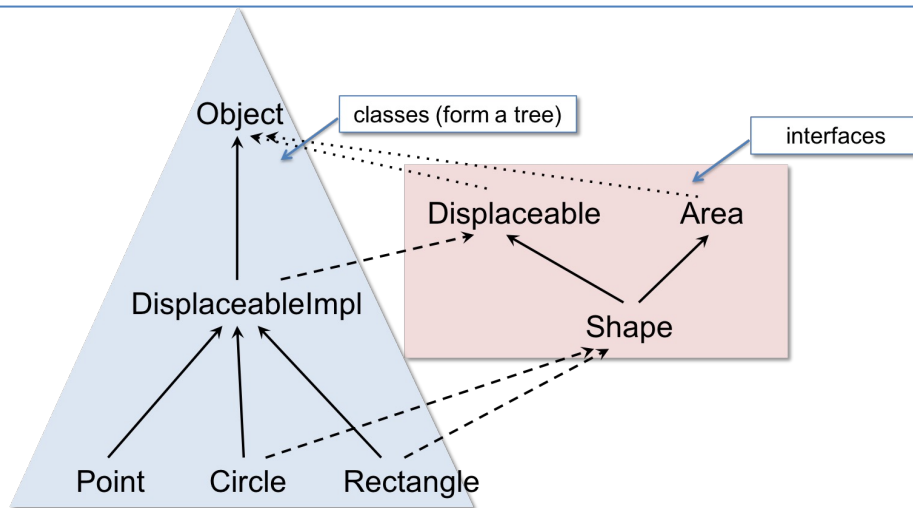
...

```
Point p = new Point(5,5);
Circle c = new Circle (p,3);
Area a1 = c; // A
```

```
--B-- y = asArea (c);
```

What type could we declare for x (in blank B)?

1. Area
2. Circle
3. Either of the above
4. Not well typed



— Extends  
- - - Implements  
..... Subtype by fiat

Area

# Inheritance and Dynamic Dispatch

When do constructors execute?

How are fields accessed?

What code runs in a method call?

What is 'this'?

# ASM refinement: The Class Table

Workspace

...

Stack

Heap

Class Table

# ASM refinement: The Class Table

```
public class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}  
  
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}
```

The class table contains:

- the code for each method,
- references to each class's parent, and
- the class's static members.

Class Table

## Object

String toString(){...}

boolean equals...

...

## Counter

extends

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

## Decr

extends

Decr(int initY) { ... }

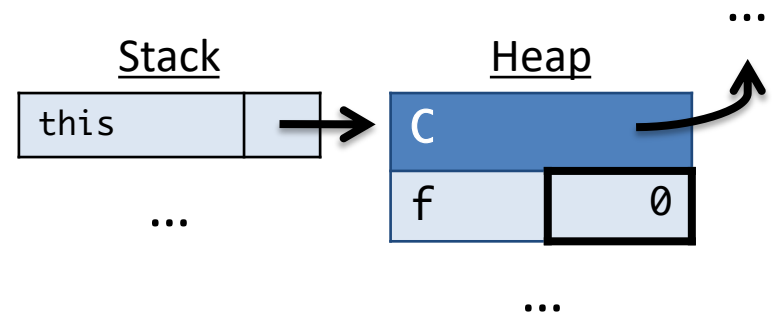
void dec(){incBy(-y);}



# this

- Inside a non-static method, the identifier `this` is an immutable reference to the object on which the method was invoked.
- References to local fields and methods have an implicit “`this.`” in front of them.

```
class C {  
    private int f;  
  
    public void copyF(C other) {  
        this.f = other.f;  
    }  
}
```



# An Example

```
public class Counter {  
    private int x;  
    public Counter () { x = 0; }  
    public void incBy(int d) { x = x + d; }  
    public int get() { return x; }  
}
```

```
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { y = initY; }  
    public void dec() { incBy(-y); }  
}
```

```
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

# ...with Explicit this and super

```
public class Counter extends Object {  
    private int x;  
    public Counter () { super(); this.x = 0; }  
    public void incBy(int d) { this.x = this.x + d; }  
    public int get() { return this.x; }  
}
```

```
public class Decr extends Counter {  
    private int y;  
    public Decr (int initY) { super(); this.y = initY; }  
    public void dec() { this.incBy(-this.y); }  
}
```

```
// ... somewhere in main:  
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

# Constructing an Object

## Workspace

```
Decr d = new Decr(2);  
d.dec();  
int x = d.get();
```

## Stack

## Heap

## Class Table

### Object

String toString(){...}

boolean equals...

...

### Counter

extends

Counter() { x = 0; }

void incBy(int d){...}

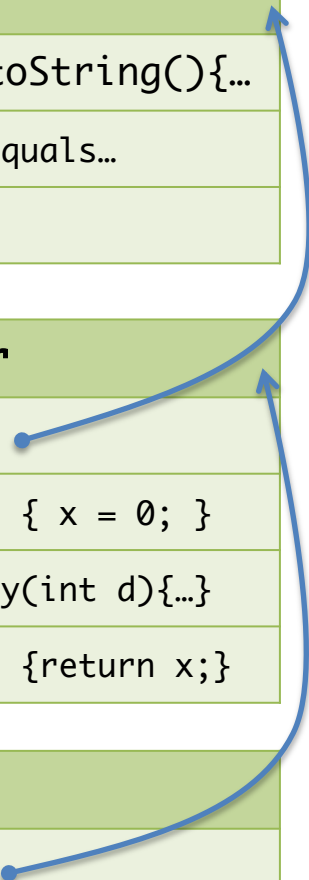
int get() {return x;}

### Decr

extends

Decr(int initY) { ... }

void dec(){incBy(-y);}



# Allocating Space on the Heap

## Workspace

```
super();  
this.y = initY;
```

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this	→
------	---

initY	2
-------	---

## Heap

Decr	
x	0
y	0

## Class Table

### Object

```
String toString(){...}  
boolean equals...  
...
```

### Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Invoking a constructor:

- allocates space for a new object in the heap
- includes slots for *all* fields of *all* ancestors in the class tree (here: *x and y*)
- creates a pointer to the class – this is the object's dynamic type
- runs the constructor body after pushing parameters and **this** onto the stack

Note: fields start with a “sensible” default

- 0 for numeric values
- null for references

# Calling super

## Workspace

```
super();  
this.y = initY;
```

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this	→
------	---

initY	2
-------	---

## Heap

Decr	
x	0
y	0

## Class Table

### Object

String toString(){...}

boolean equals...

...

### Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

### Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Call to super:

- The constructor (implicitly) calls the super constructor
- Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new **this** pointer.

# Abstract Stack Machine

## Workspace

```
super();  
this.x = 0;
```

(Running Object's default  
constructor omitted.)

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

```
_  
this.y = initY;
```

this

## Heap

Decr

x

0

y

0

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

# Assigning to a Field

## Workspace

```
this.x = 0;
```

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

```
_  
this.y = initY;
```

this

## Heap

Decr

x

0

y

0

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Assignment into the `this.x` field goes in two steps:

- look up the value of `this` in the stack
- write to the "x" slot of that object.



# Assigning to a Field

## Workspace

`.X = 0;`

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

```
_  
this.y = initY;
```

this

## Heap

Decr

x

y

0

0

## Class Table

### Object

String toString(){...}

boolean equals...

...

### Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

### Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Assignment into the `this.x` field goes in two steps:

- look up the value of `this` in the stack
- write to the "x" slot of that object.

# Done with the call

## Workspace

;

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

```
_  
this.y = initY;
```

this

## Heap

Decr

x

0

y

0

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Done with the call to “super”, so  
pop the stack to the previous  
workspace.

# Continuing

## Workspace

this.y = initY;

Continue in the Decr class's constructor.

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

## Heap

Decr

x

0

y

0

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

# Abstract Stack Machine

## Workspace

```
this.y = 2;
```

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

## Heap

Decr

x

0

y

0

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

# Assigning to a field

## Workspace

```
this.y = 2;
```

## Stack

```
Decr d = ...;  
d.dec();  
int x = d.get();
```

this	
initY	2

## Heap

Decr	
x	0
y	2

## Class Table

### Object

```
String toString(){...}  
boolean equals...  
...
```

### Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Assignment into the `this.y` field.

(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity...)

# Done with the call

## Workspace

;

## Stack

```
Decr d = _;  
d.dec();  
int x = d.get();
```

this

initY

2

## Heap

Decr

x

0

y

2

## Class Table

**Object**

String toString(){...}

boolean equals...

...

**Counter**

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

**Decr**

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Done with the call to the Decr constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the this pointer).

# Returning the Newly Constructed Object

## Workspace

```
Decr d = .;  
d.dec();  
int x = d.get();
```

## Stack

## Heap

Decr	
x	0
y	2

## Class Table

### Object

String toString(){...}  
boolean equals...  
...

### Counter

extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}

### Decr

extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}

Continue executing the program.

# Allocating a local variable

## Workspace

```
d.dec();  
int x = d.get();
```

## Stack



## Heap



## Class Table

### Object

```
String toString(){...}  
boolean equals...  
...
```

### Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Allocate a stack slot for the local variable **d**. Note that it's mutable... (bold box in the diagram).

Aside: since, by default, fields and local variables are mutable in Java, we sometimes omit the bold boxes and just assume the contents can be modified.



# Dynamic Dispatch: Finding the Code

## Workspace

```
d.dec();  
int x = d.get();
```

## Stack



## Heap



## Class Table

### Object

```
String toString(){...  
boolean equals...  
...
```

### Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Invoke the **dec** method on the object. The code is found by “pointer chasing” through the class hierarchy.

This is an example of *dynamic dispatch*: Which code is run depends on the dynamic class of the object. (In this case, **Decr**.)

Search through the methods of the **Decr**, class trying to find one called **dec**.

# Dynamic Dispatch: Finding the Code

## Workspace

```
this.incBy(-this.y);
```

Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

## Stack

d	•
---	---

```
int x = d.get();
```

this	•
------	---

## Heap

Decr	
x	0
y	2

## Class Table

### Object

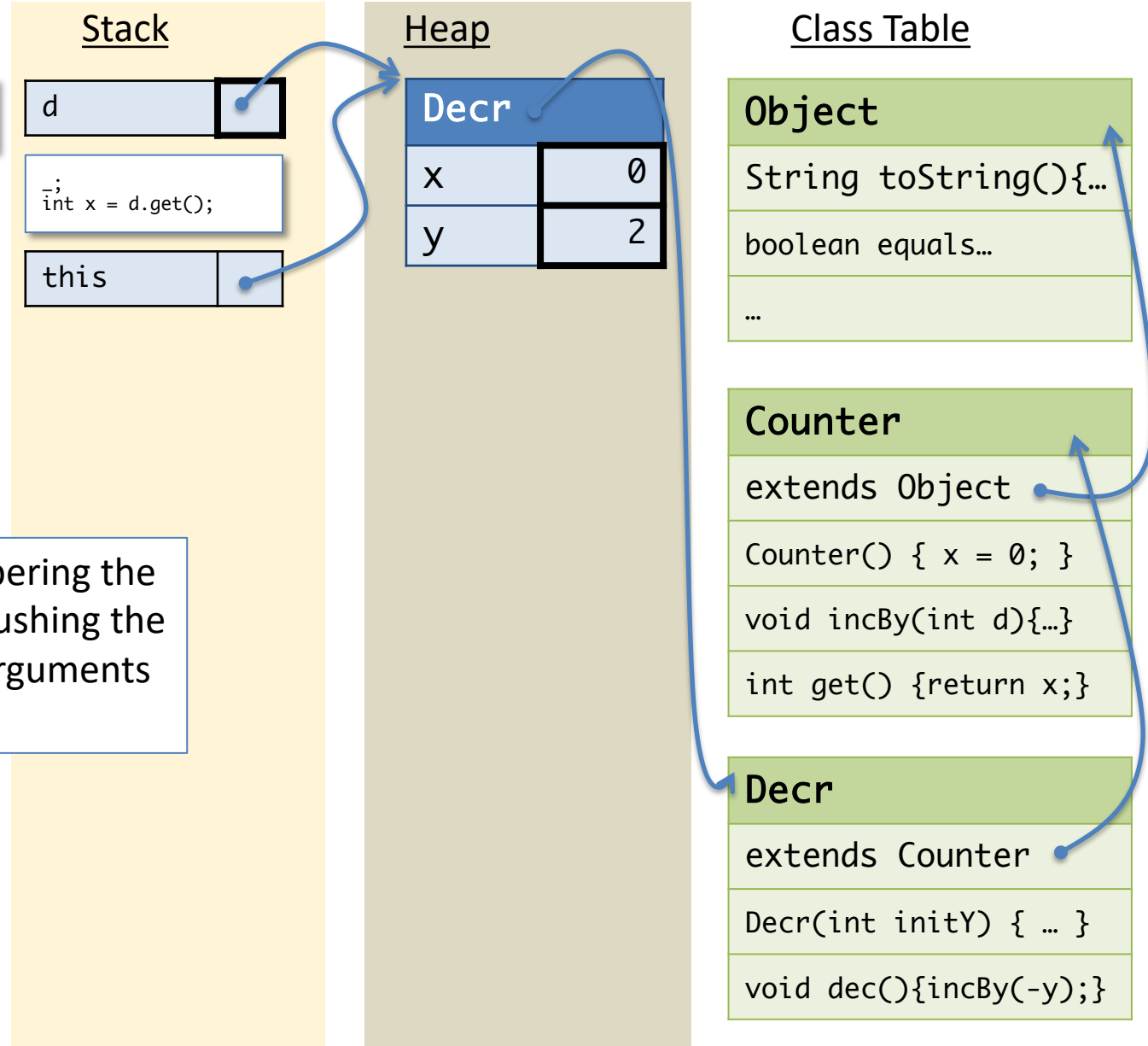
```
String toString(){...}  
boolean equals...  
...
```

### Counter

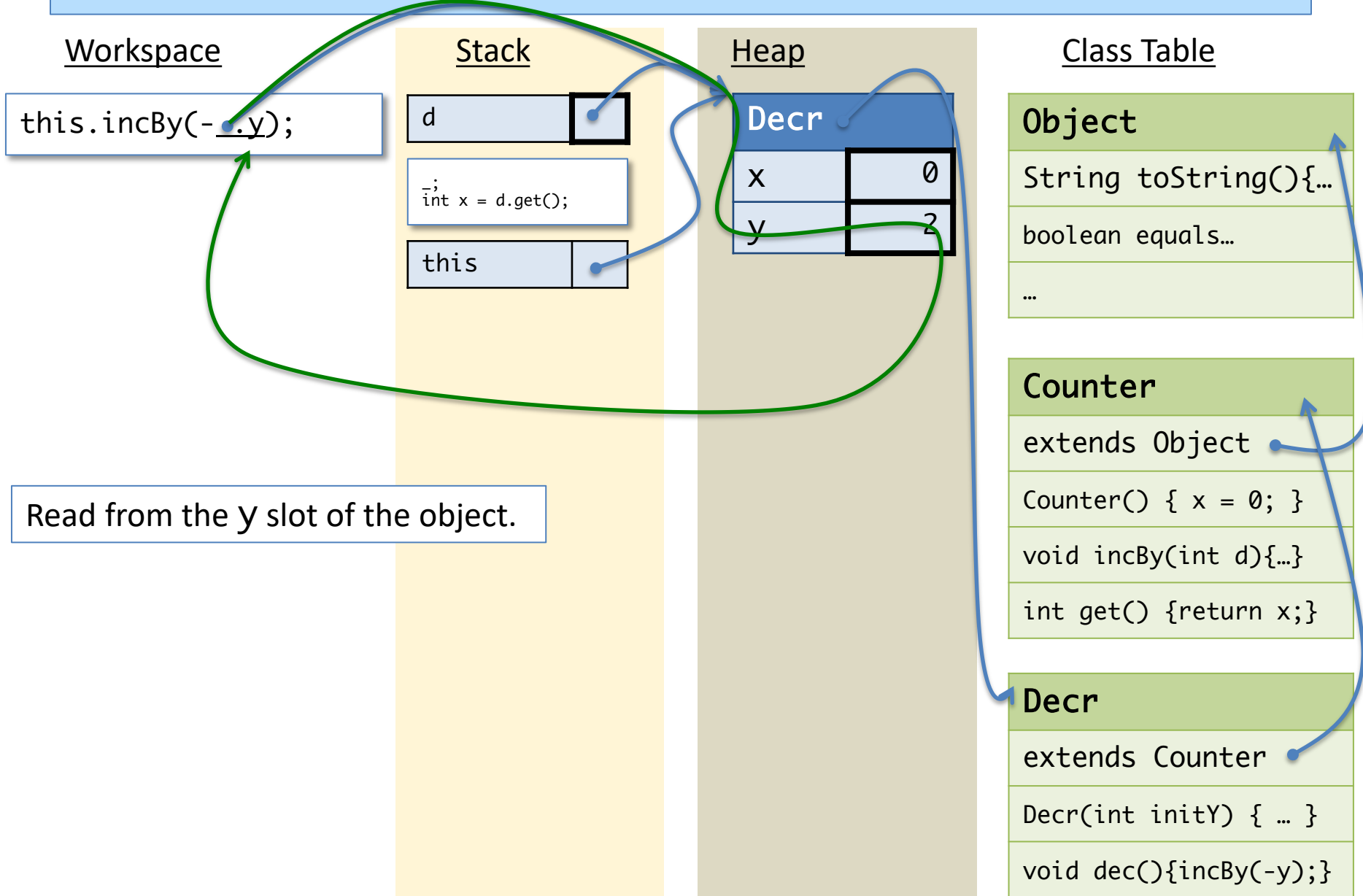
```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```



# Reading a Field's Contents



# Dynamic Dispatch, Again

## Workspace

```
d.incBy(-2);
```

## Stack

d

```
int x = d.get();
```

this

## Heap

Decr

x 0

y 2

## Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Invoke the `incBy` method on the object via dynamic dispatch.

In this case, the `incBy` method is *inherited* from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.

The search is guaranteed to succeed – Java's static type system ensures this.

Search through the methods of the `Decr` class looking for one called `incBy`. If the search fails, recursively search the parent classes.

# Running the body of incBy

## Workspace

```
this.x = this.x + d;
```



```
this.x = -2;
```

## Stack

d

```
int x = d.get();
```

this

-;

d -2

this

## Heap

Decr

x -2

y 2

## Class Table

### Object

```
String toString(){...}
```

```
boolean equals...
```

```
...
```

### Counter

```
extends Object
```

```
Counter() { x = 0; }
```

```
void incBy(int d){...}
```

```
int get() {return x;}
```

### Decr

```
extends Counter
```

```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

It takes a few steps...

Body of incBy:

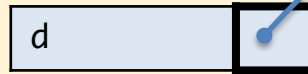
- reads this.x
- looks up d
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

# After a few more steps...

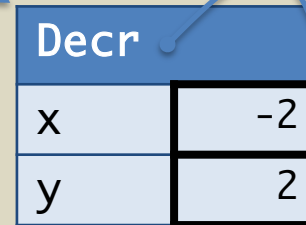
## Workspace

```
int x = d.get();
```

## Stack



## Heap



## Class Table

### Object

```
String toString(){...}  
boolean equals...  
...
```

### Counter

```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

### Decr

```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Now use dynamic dispatch to invoke the **get** method for d. This involves searching up the class hierarchy again...

# After yet a few more steps...

## Workspace

Done! (Phew!)

## Stack

d	
x	-2

## Heap

Decr	
x	-2
y	2

## Class Table

### Object

String toString(){...}  
boolean equals...  
...

### Counter

extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}

### Decr

extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}

# Summary: `this` and dynamic dispatch

- When object's method is invoked, as in `O.m()`, the code that runs is determined by `O`'s *dynamic* class.
  - The dynamic class, represented as a pointer into the class table, is included in the object structure in the heap
  - If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table
  - This process of *dynamic dispatch* is the heart of OOP!
- Once the code for `m` has been determined, a binding for `this` is pushed onto the stack.
  - The `this` pointer is used to resolve field accesses and method invocations inside the code.



# Static members and the Java ASM

# Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

You can do a static assignment to initialize a static field.

```
class C {  
    public static int x = 23;  
    public static int someMethod(int y) { return C.x + y; }  
    public static void main(String args[]) {  
        ...  
    }  
}
```

```
// Elsewhere:  
C.x = C.x + 1;  
C.someMethod(17);
```

Access to the static member uses the class name  
`C.x` or `C.foo()`

Based on your understanding of 'this', is it possible to refer to 'this' in a static method?

1. No
2. Yes
3. I'm not sure

# Example of Statics

- The `java.lang.Math` library provides static fields/methods for many common arithmetic operations:
- `Math.PI == 3.141592653589793`
- `Math.sin`, `Math.cos`
- `Math.sqrt`
- `Math.pow`
- etc.

# Class Table Associated with C

- The class table entry for C has a field slot for x.
- Updates to C.x modify the contents of this slot: C.x = 17;

C	
extends Object	
static x	23
static int someMethod(int y) { return x + y; }	
static void main(String args[]) {...}	

- A static field is a *global* variable
  - There is only one heap location for it (in the class table)
  - Modifications to such a field are visible everywhere the field is
    - if the field is public, this means *everywhere*
  - Use with care!

# Static Methods (Details)

- Static methods do *not* have access to a `this` pointer
  - Why? There isn't an instance to dispatch through!
  - Therefore, static methods may only directly call other static methods.
  - Similarly, static methods can only directly read/write static fields.
  - Of course, a static method can create instance of objects (via `new`) and then invoke methods on those objects.
- Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
  - e.g. `o.someMethod(17)` where `someMethod` is static