Programming Languages and Techniques (CIS1200)

Lecture 25

Java ASM, dynamic dispatch

Chapters 23 & 24

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

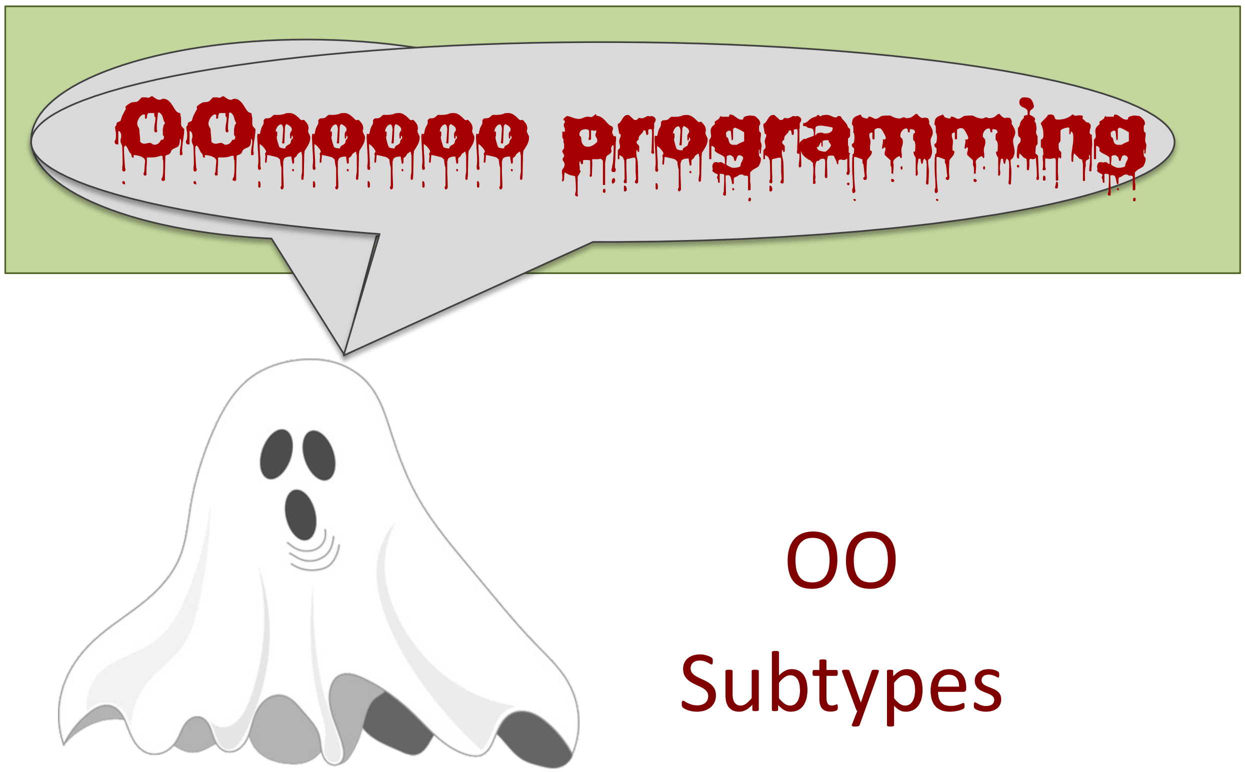
The Java Abstract Stack Machine

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 (today)*



OO

Subtypes

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

Subtyping relationships are **explicitly declared** in Java

Key Idea: Liskov's *Substitution Principle**

If B is a subtype of A, then an object of type A may be replaced with an object of type B anywhere an A 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.

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

supertype of Circle

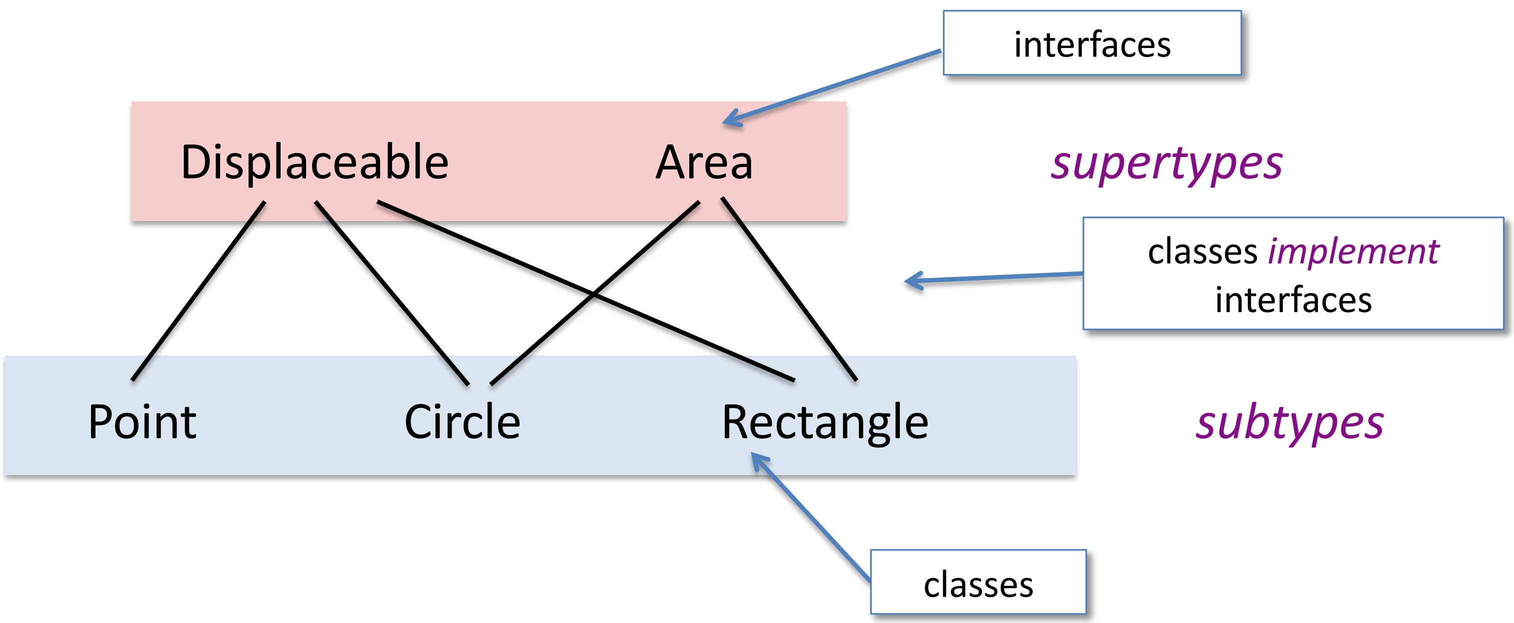
```
// in class C
public static void leapIt(Displaceable a) { a.move(10,10); }

C.leapIt(new Circle (new Point(2,3), 1));
```

subtype of Displaceable

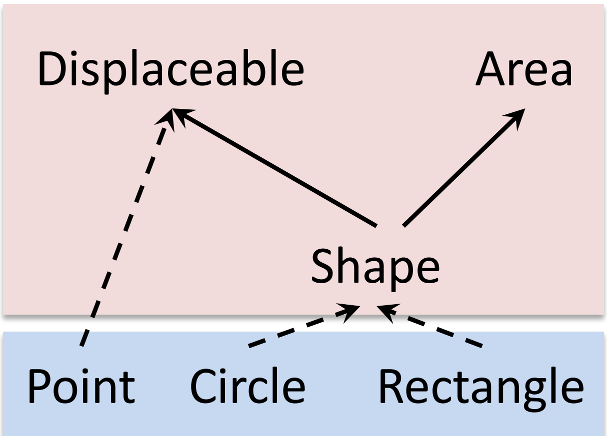
Subtypes and Supertypes

- A class that implements an interface declares that the class is a subtype of the interface



Types can have many *different* supertypes / subtypes

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.
  - Should reflect an “is a” relationship between objects (e.g., a Car *is a* Vehicle)

```
public class Vehicle {
   private int x; private int y;
   public void go() { … update x … }
}

public class ElectricCar extends Vehicle {
   public void charge() { … }
}
```

Simple Inheritance

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

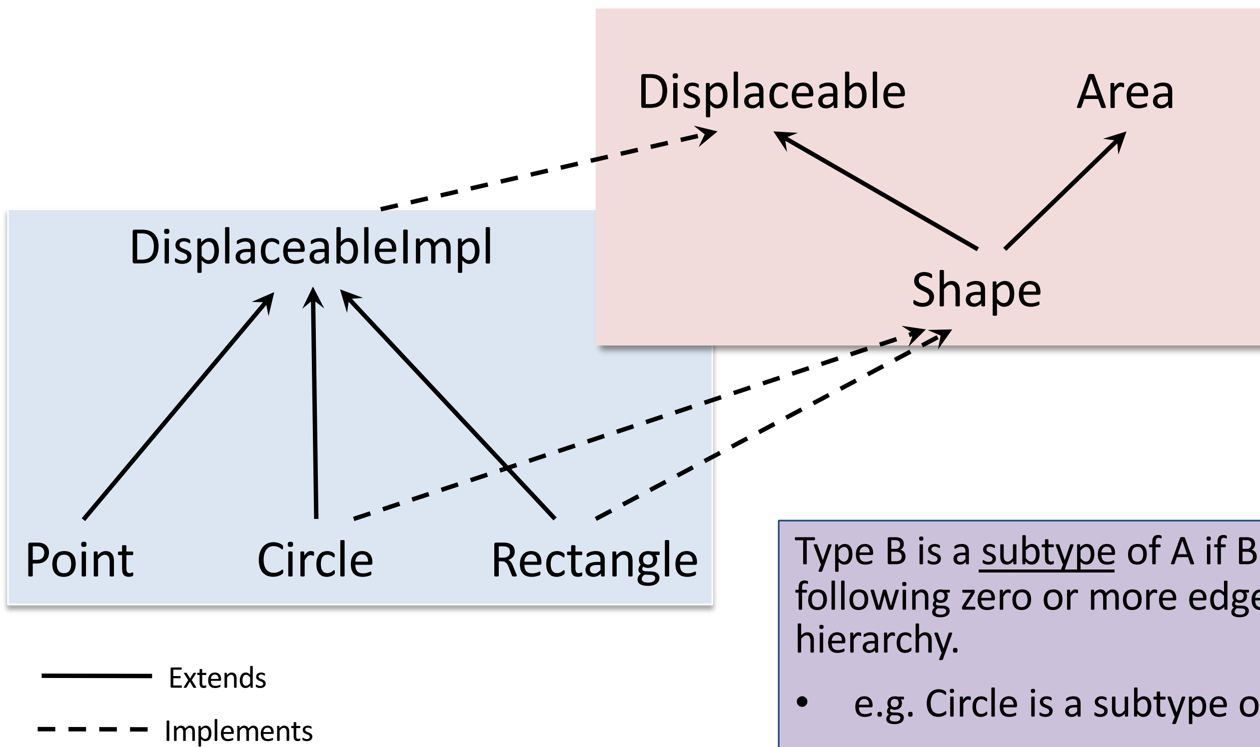
```
public class DisplaceableImpl implements Displaceable {
   private int x; private int y;
   public DisplaceableImpl(int x, int y) { … }
   public int getX() { return x;}
   public int getY() { return y; }
   public void move(int dx, int dy) { x += dx; y += dy; }
}
```

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*

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)

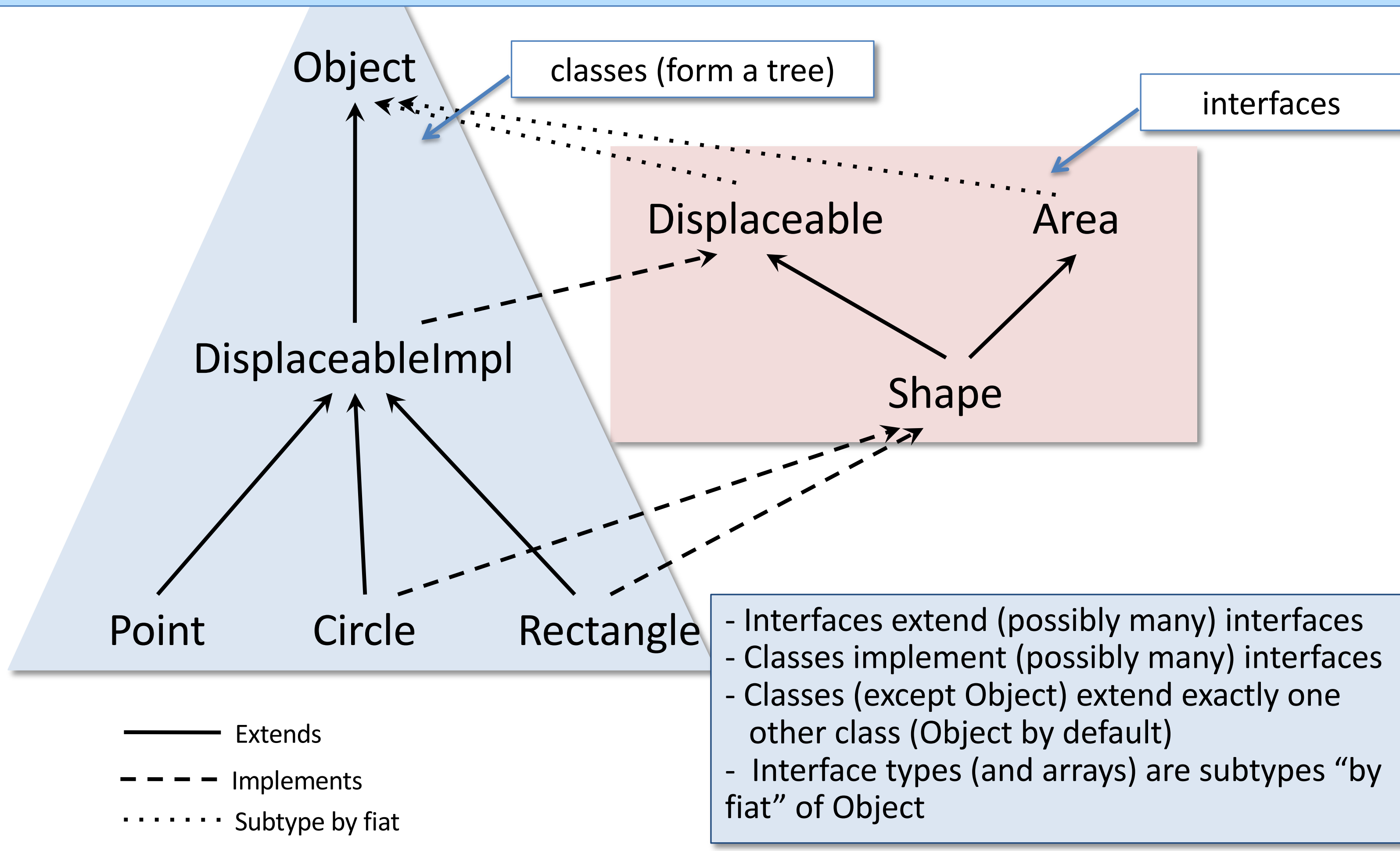
```
public Circle(Point pt, int radius) {
     super(pt.getX(),pt.getY());
     this.radius = radius;
  }
```

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



- 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 using it is part of a well-known "contract" of the design: easy to violate Liskov substitution principle

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?

0

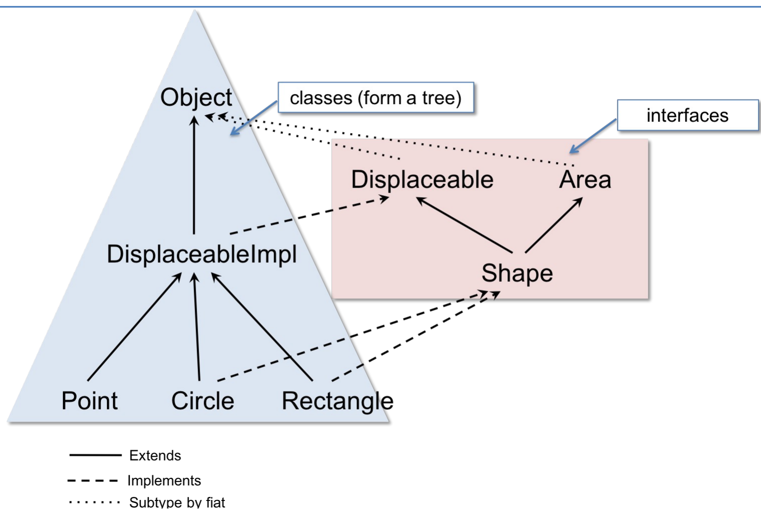
Area	0%
Circle	0%
None of the above	0%
Not well typed	0%

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?

0

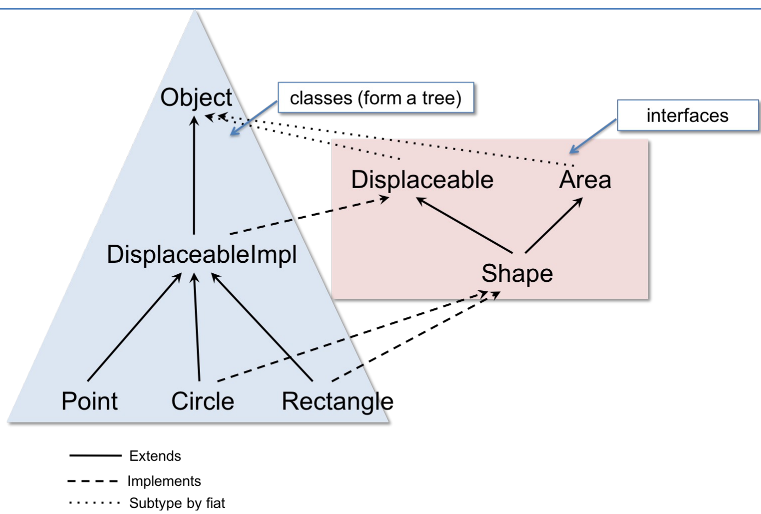
Area	0%
Circle	0%
None of the above	0%
Not well typed	0%

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)?

0

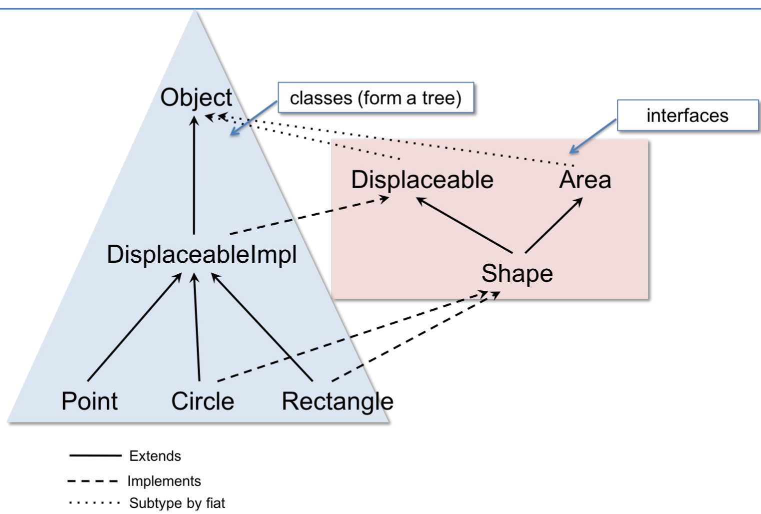
Area	0%
Circle	0%
None of the above	0%
Not well typed	0%

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

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

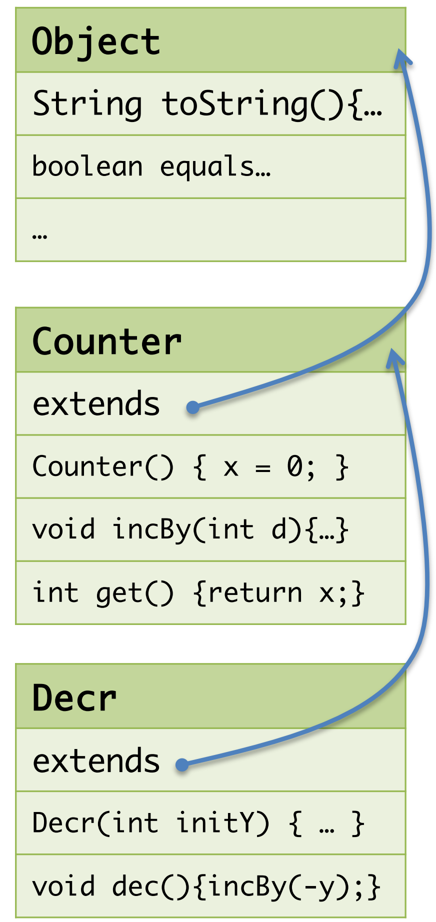
```java
public class Counter {
  private int x;
  public Counter () { x = 0; }
  public void incBy(int d) { x = x + d; }
  public int get() { return x; }
}

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

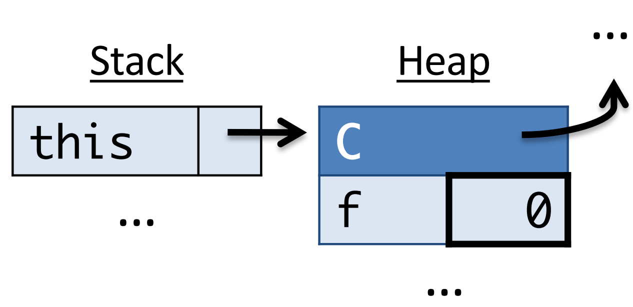


`this`

- 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

<u>Workspace</u>

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

<u>Stack</u>

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

this	
initY	2

<u>Heap</u>

Decr	
x	0
y	0

<u>Class Table</u>

Object
String toString(){…
boolean equals…
…

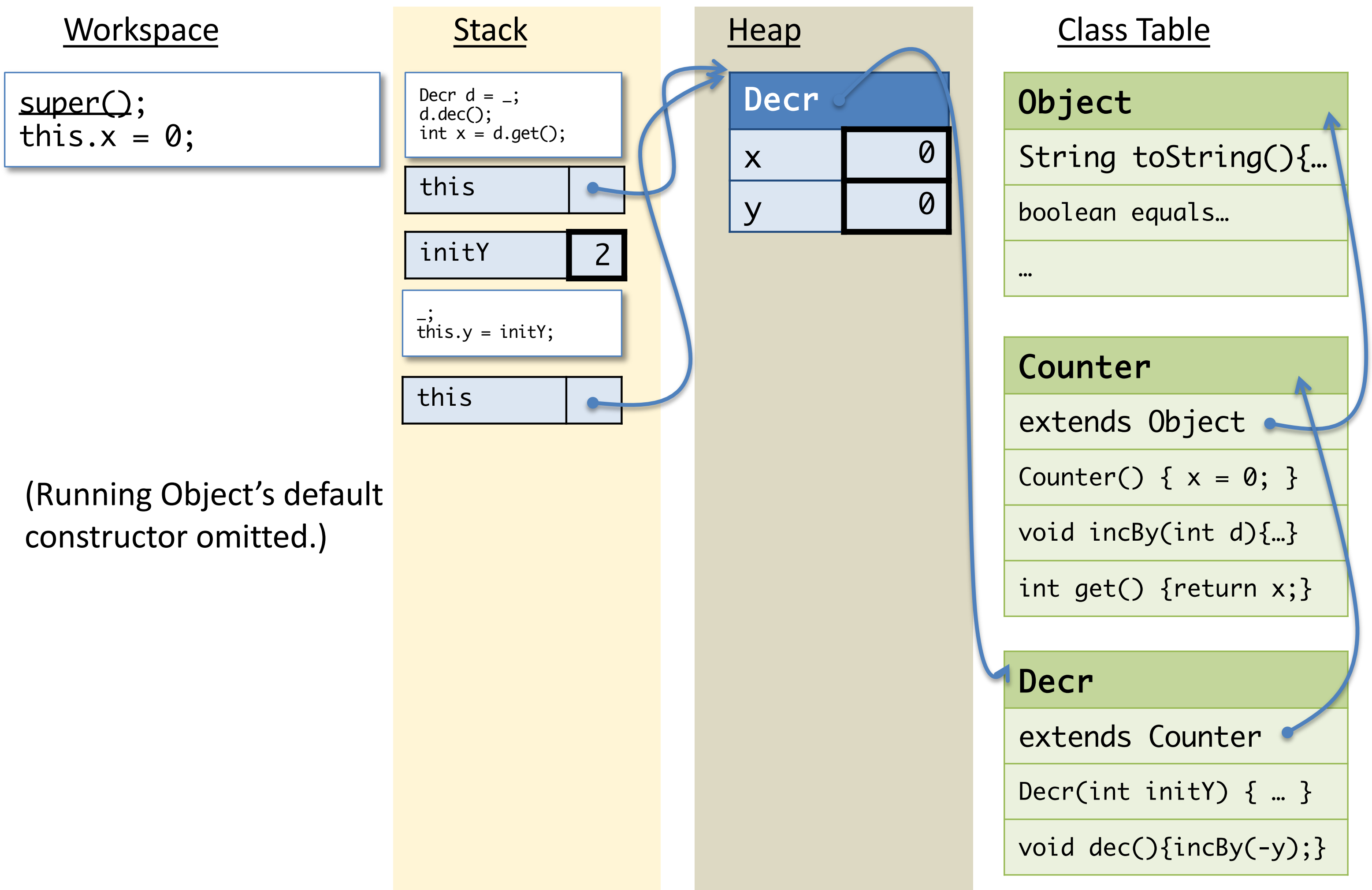
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}

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

Call to `super`:

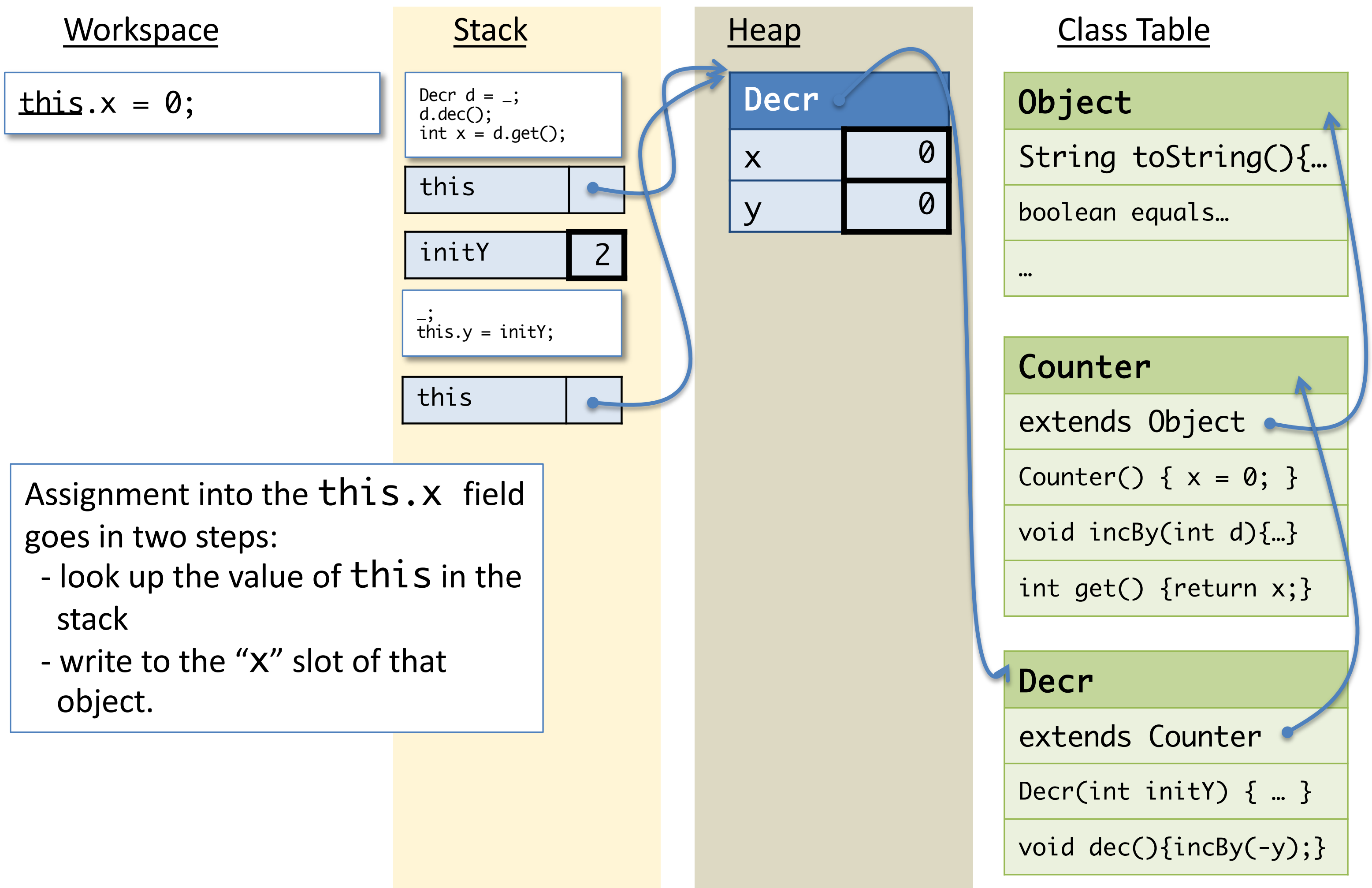
- The constructor (implicitly) calls the super constructor
- Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new `this` pointer.

Abstract Stack Machine



(Running Object's default constructor omitted.)

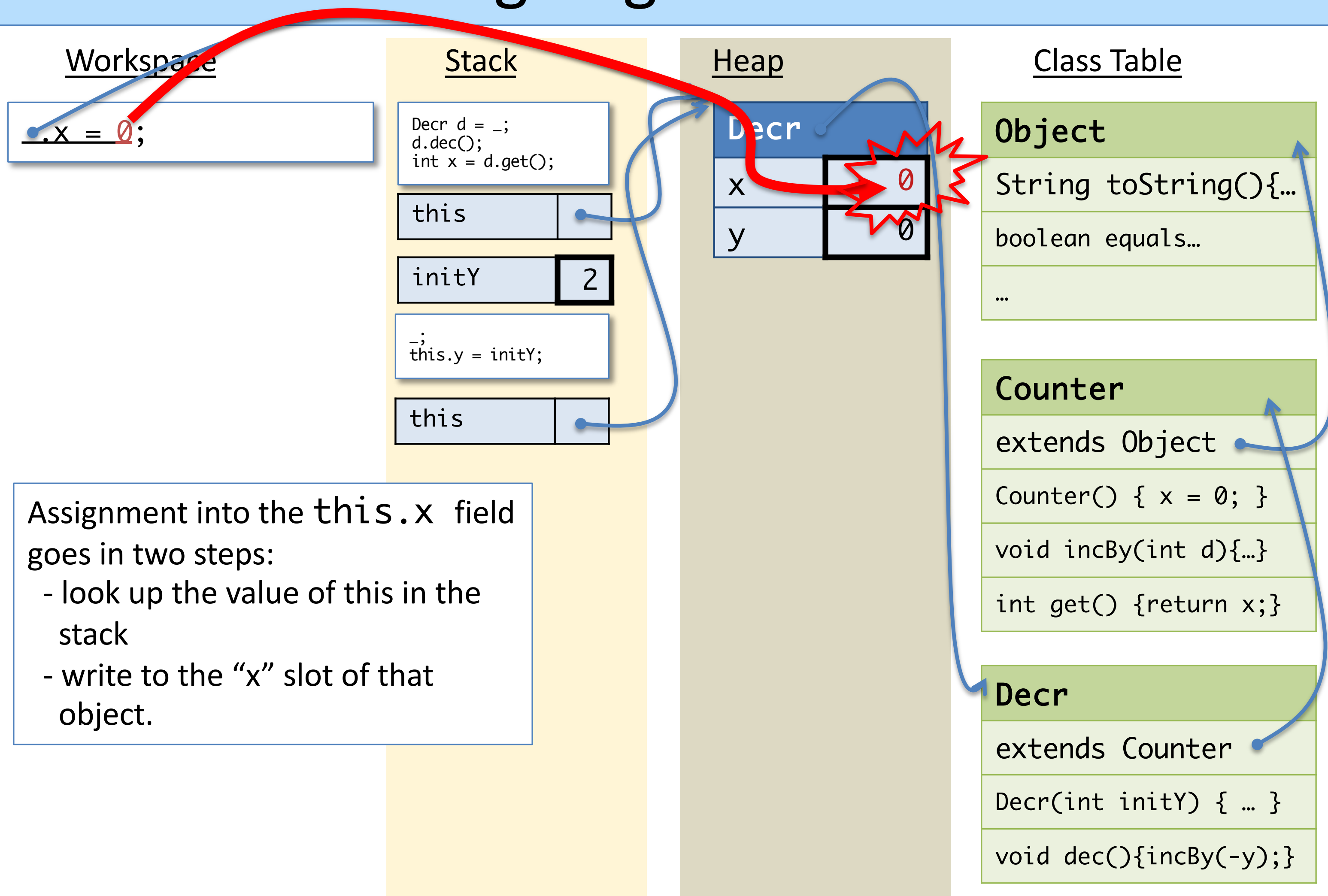
Assigning to a Field



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.

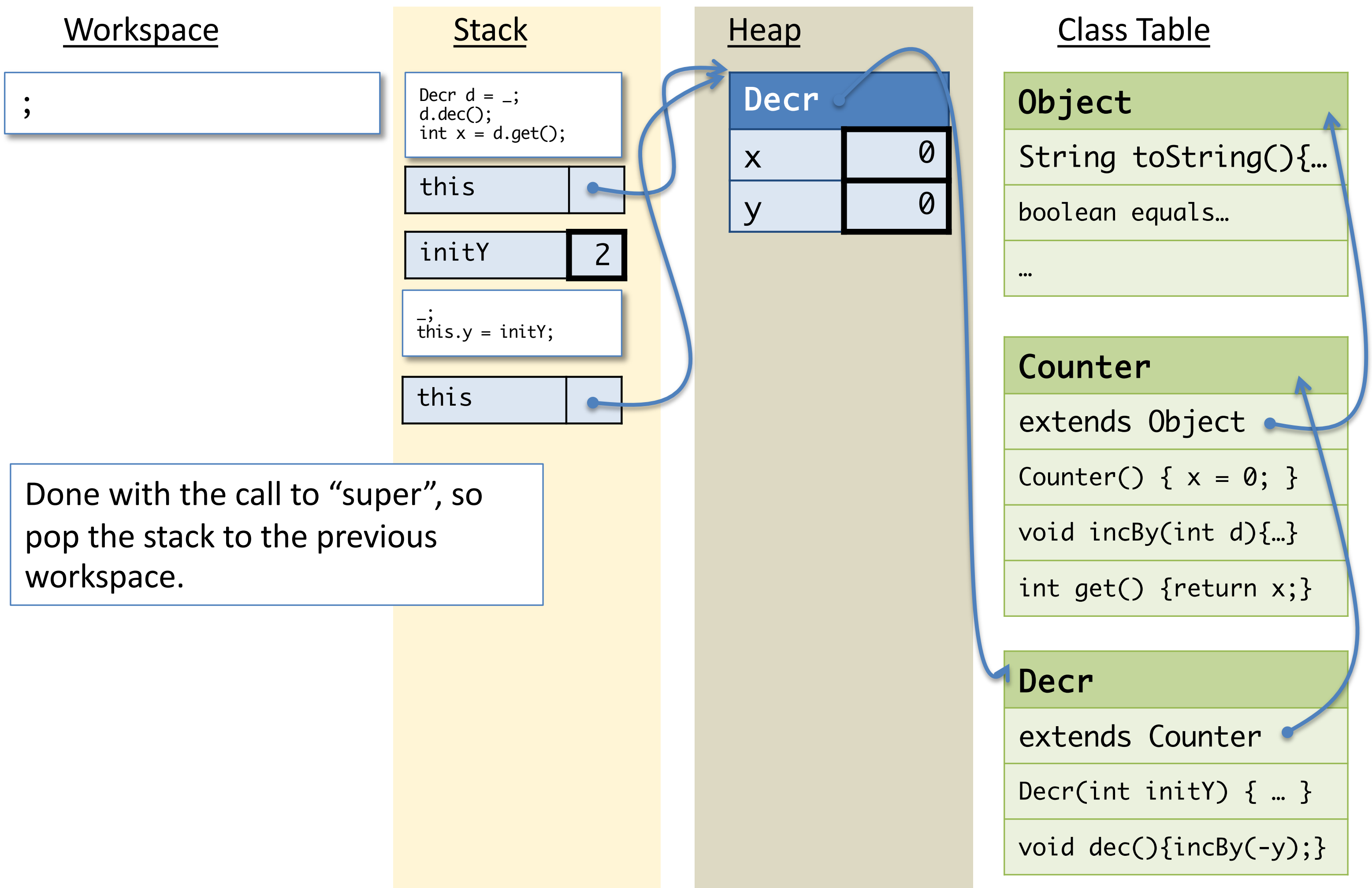
Assigning to a Field



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

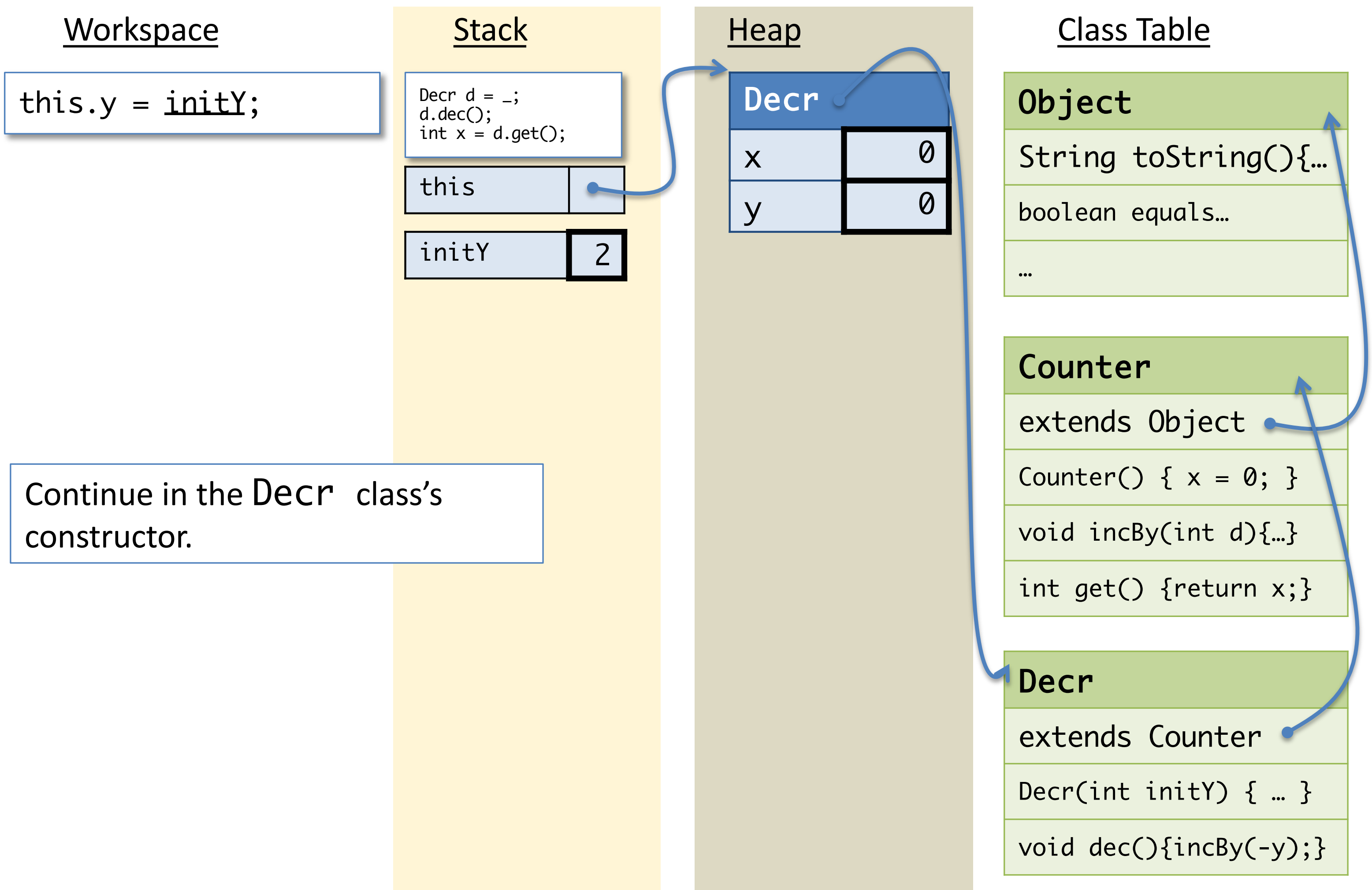
- look up the value of this in the stack
- write to the “x” slot of that object.

Done with the call



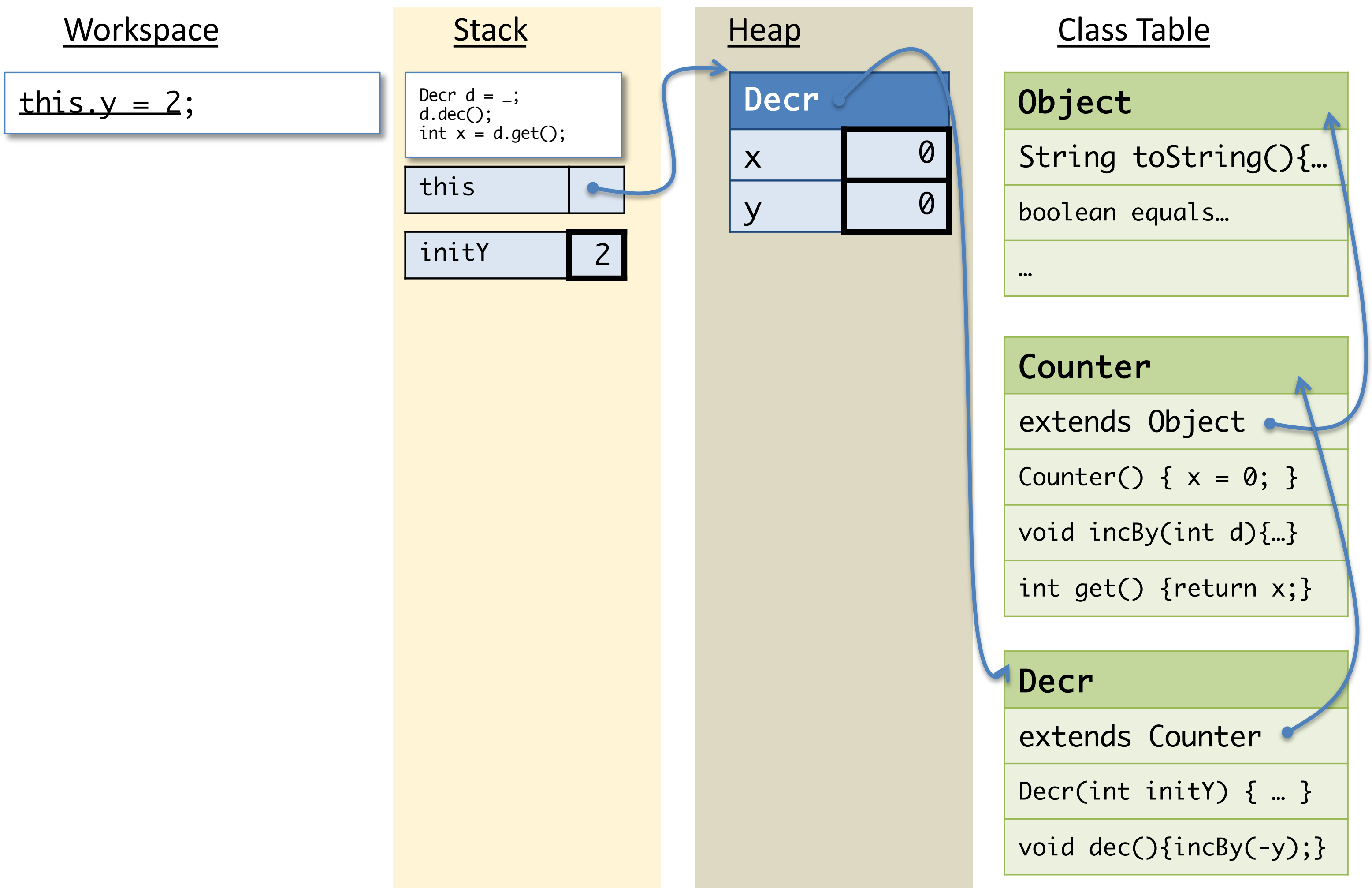
Done with the call to "super", so pop the stack to the previous workspace.

Continuing

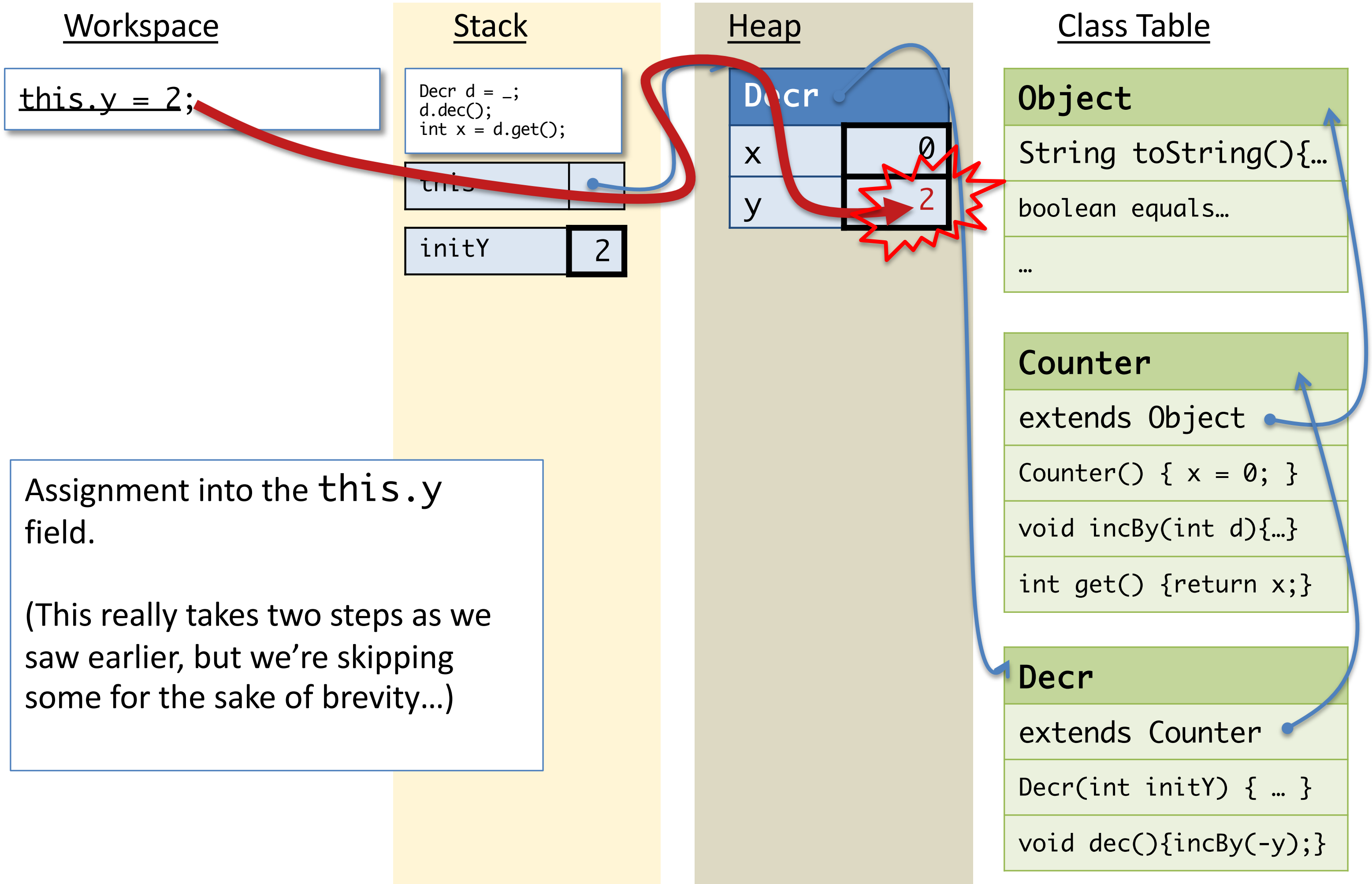


Workspace
this.y = initY;
Stack
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
Heap
Decr
x
0
y
0
Class Table
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Continue in the Decr class's constructor.

Abstract Stack Machine



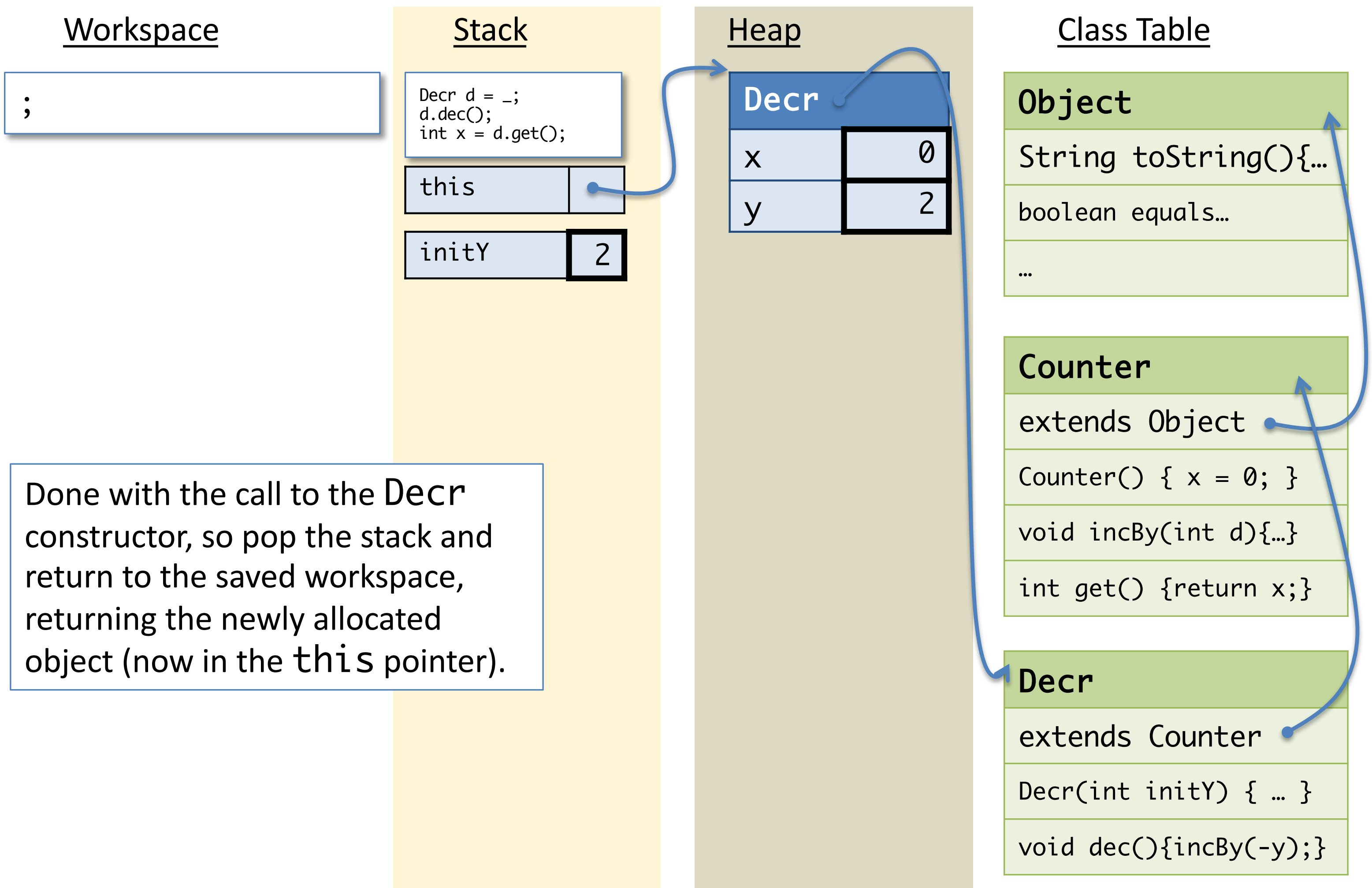
Assigning to a field



Assignment into the `this.y` field.

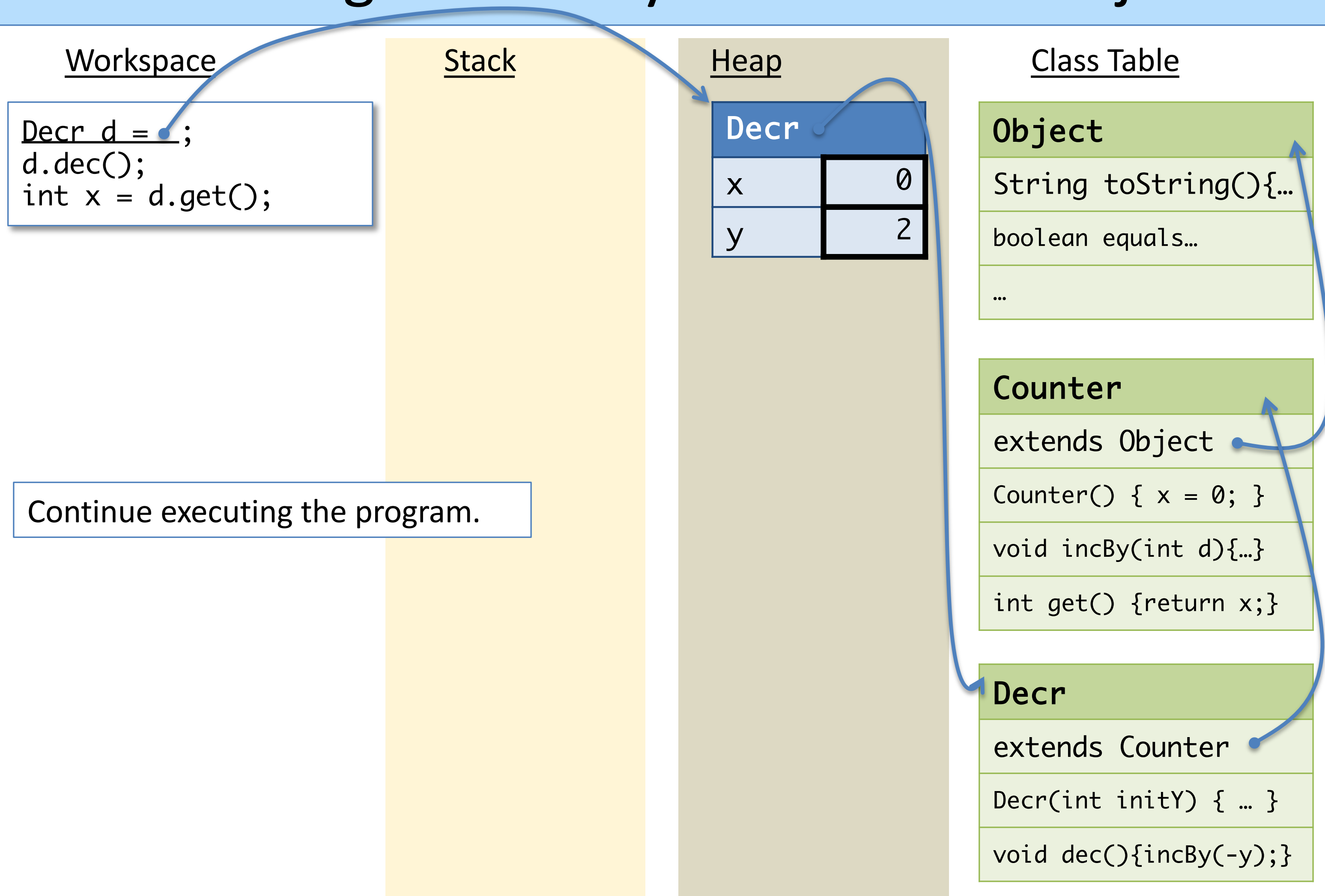
(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity…)

Done with the call

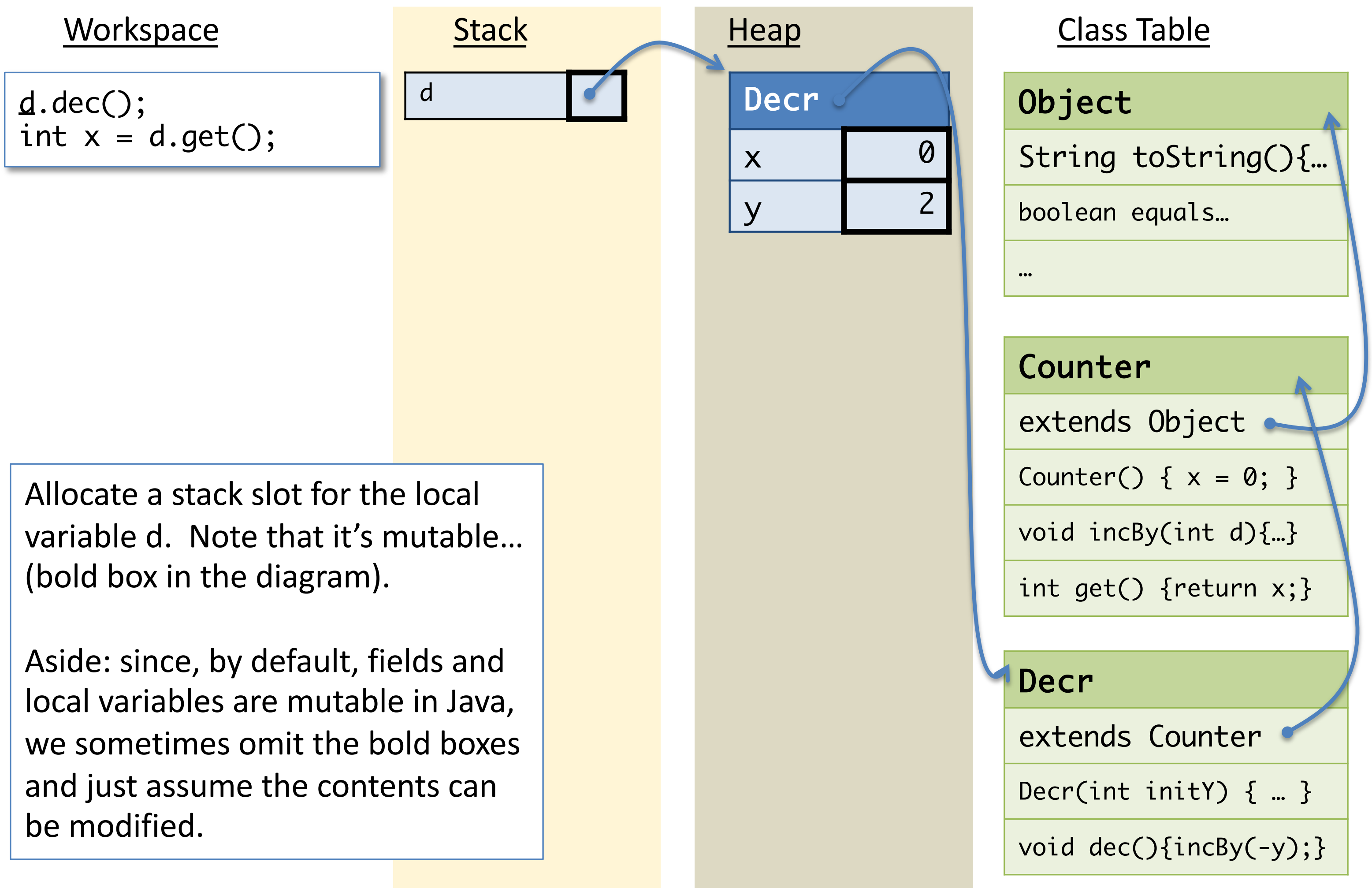


Done with the call to the `Decr` constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the `this` pointer).

Returning the Newly Constructed Object



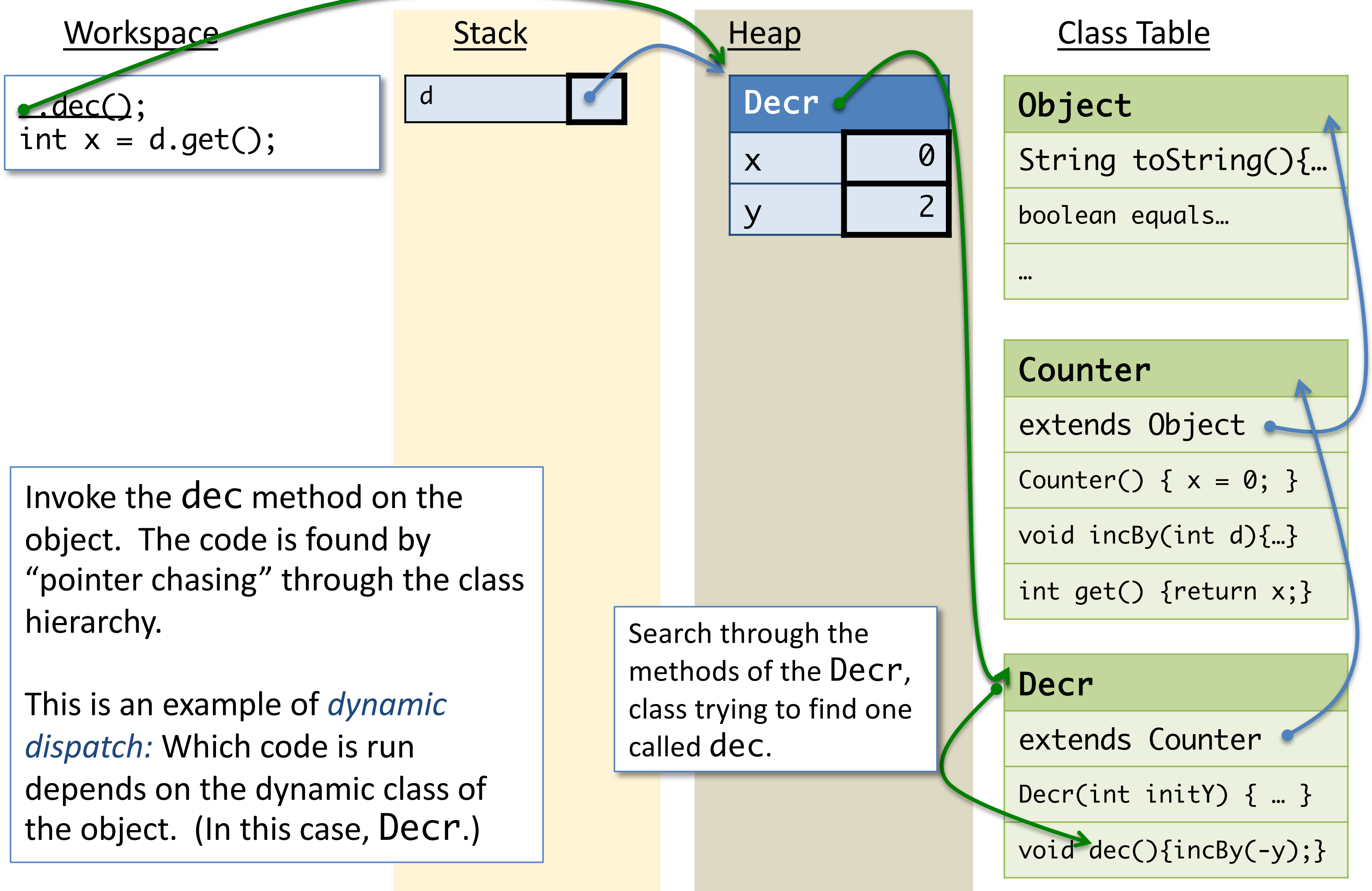
Allocating a local variable



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

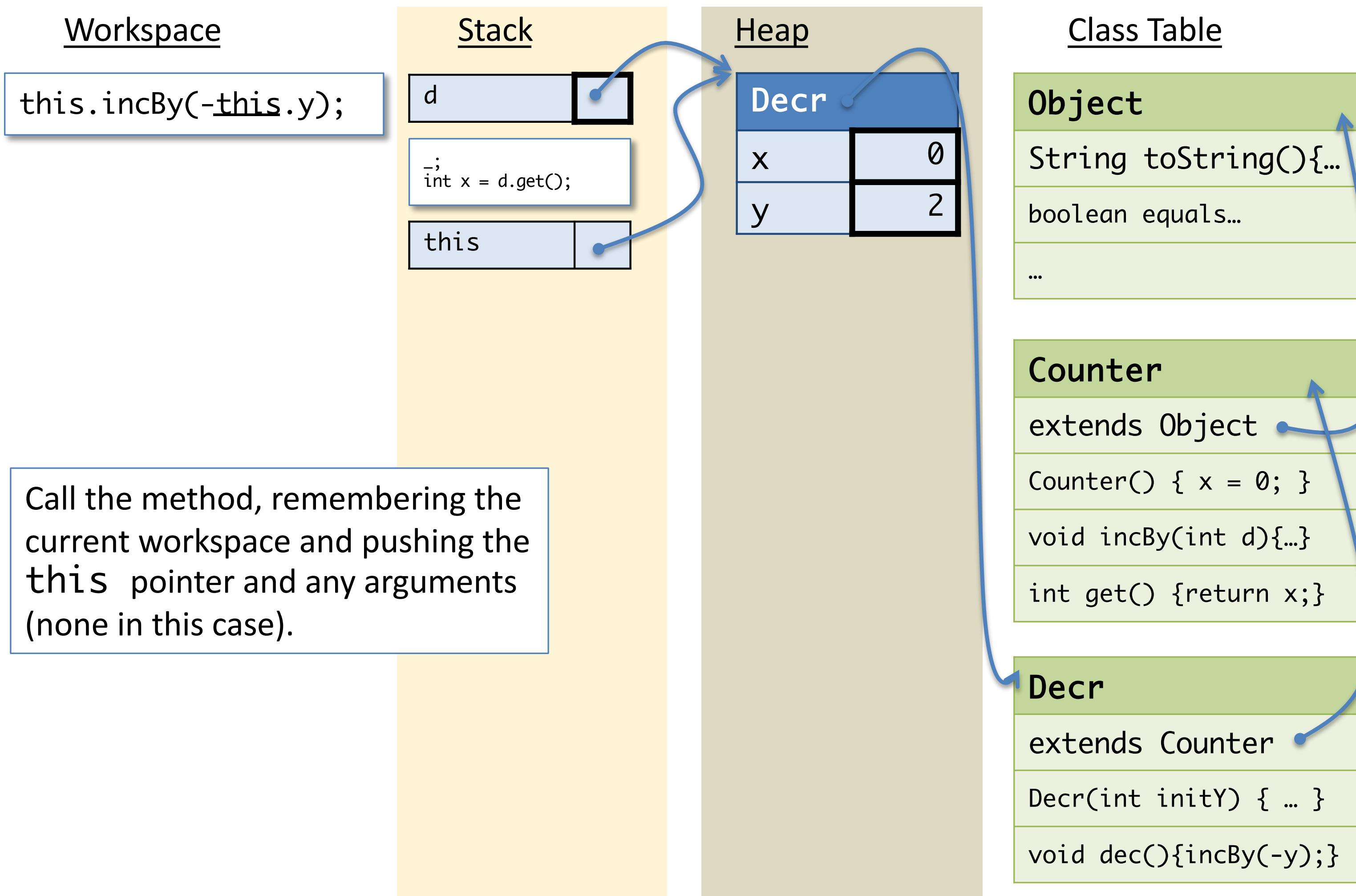


Invoke the dec method on the object. The code is found by "pointer chasing" through the class hierarchy.

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

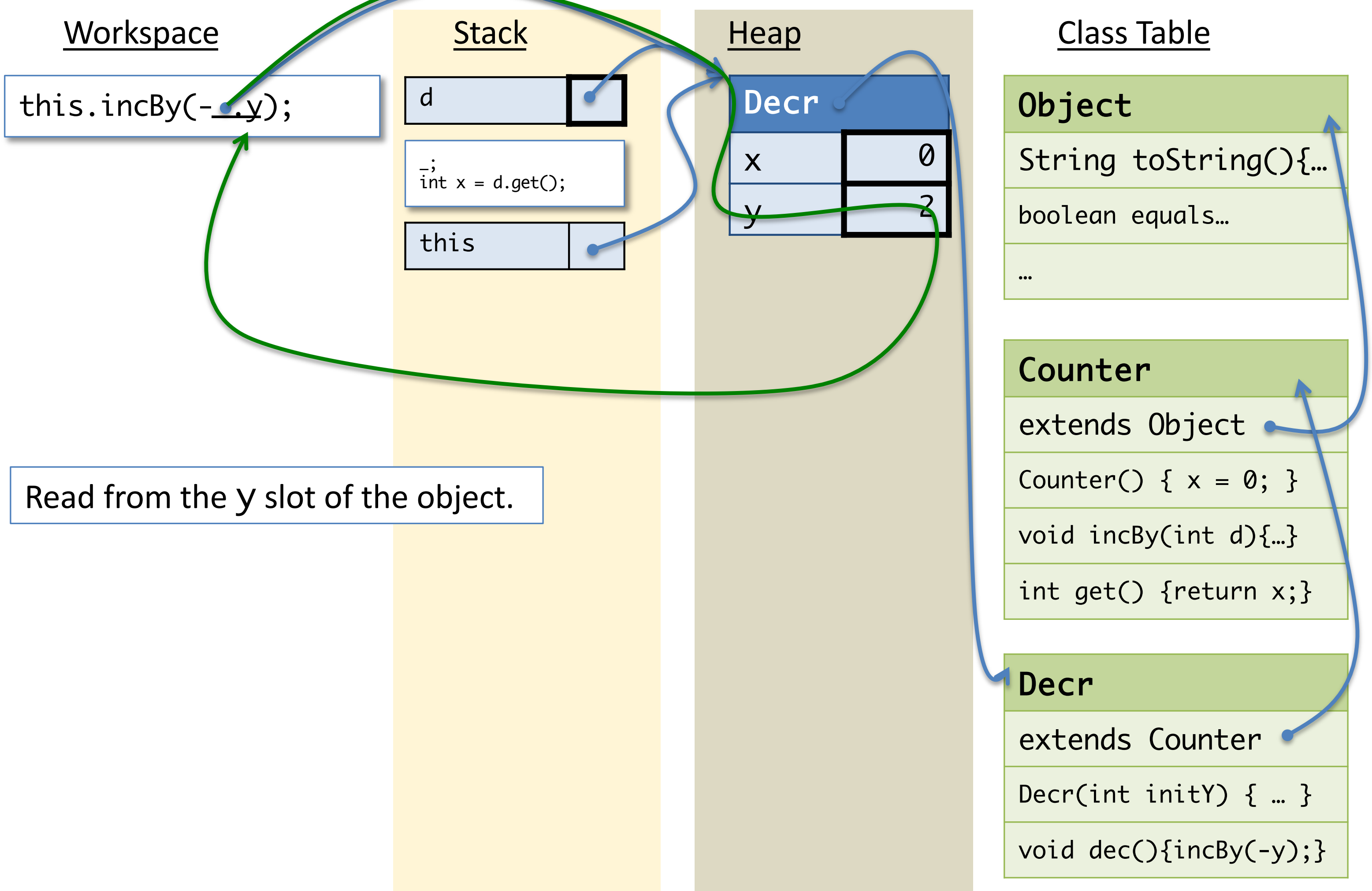
Search through the methods of the Decr, class trying to find one called dec.

Dynamic Dispatch: Finding the Code

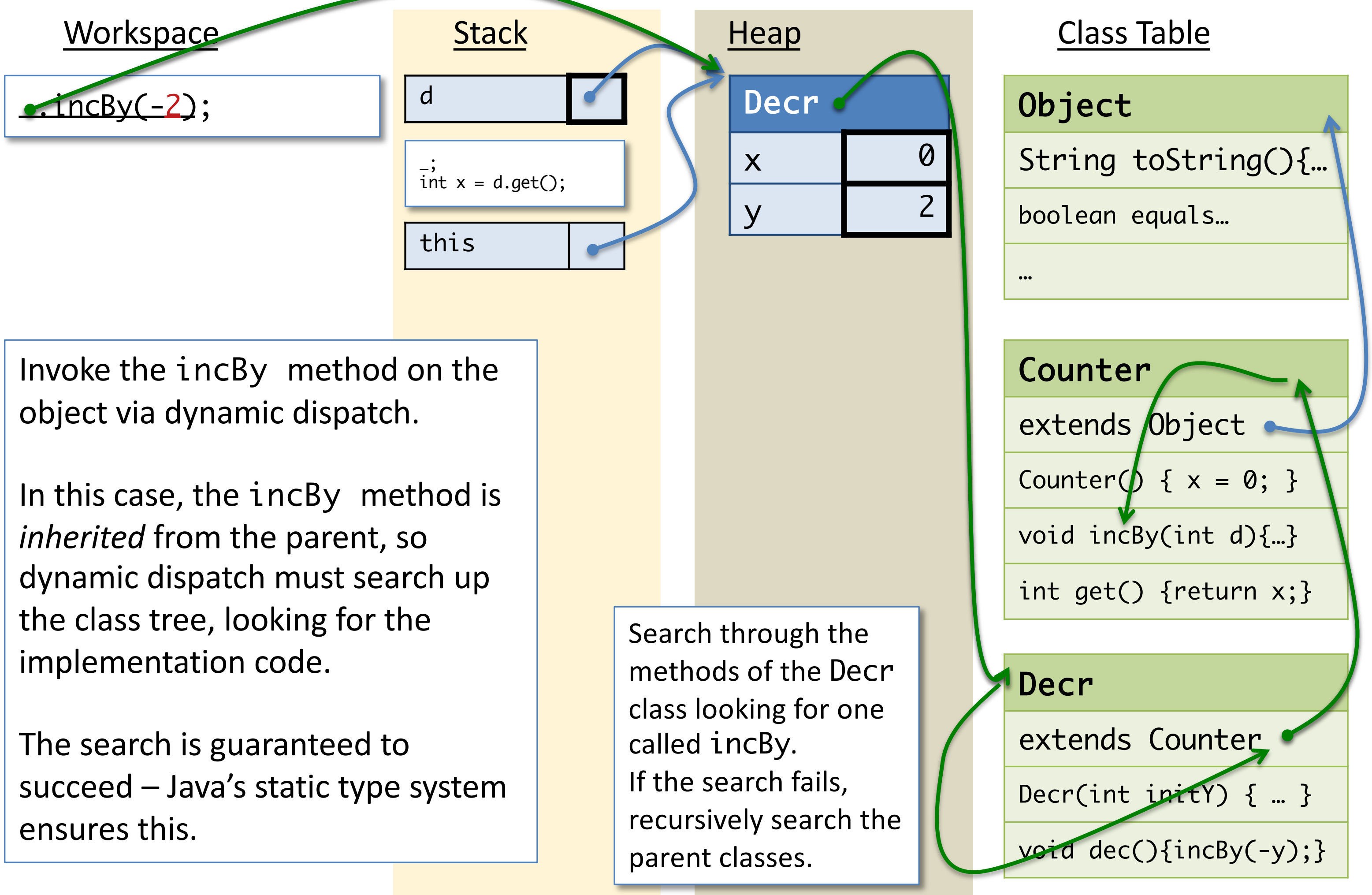


Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

Reading a Field's Contents



Dynamic Dispatch, Again



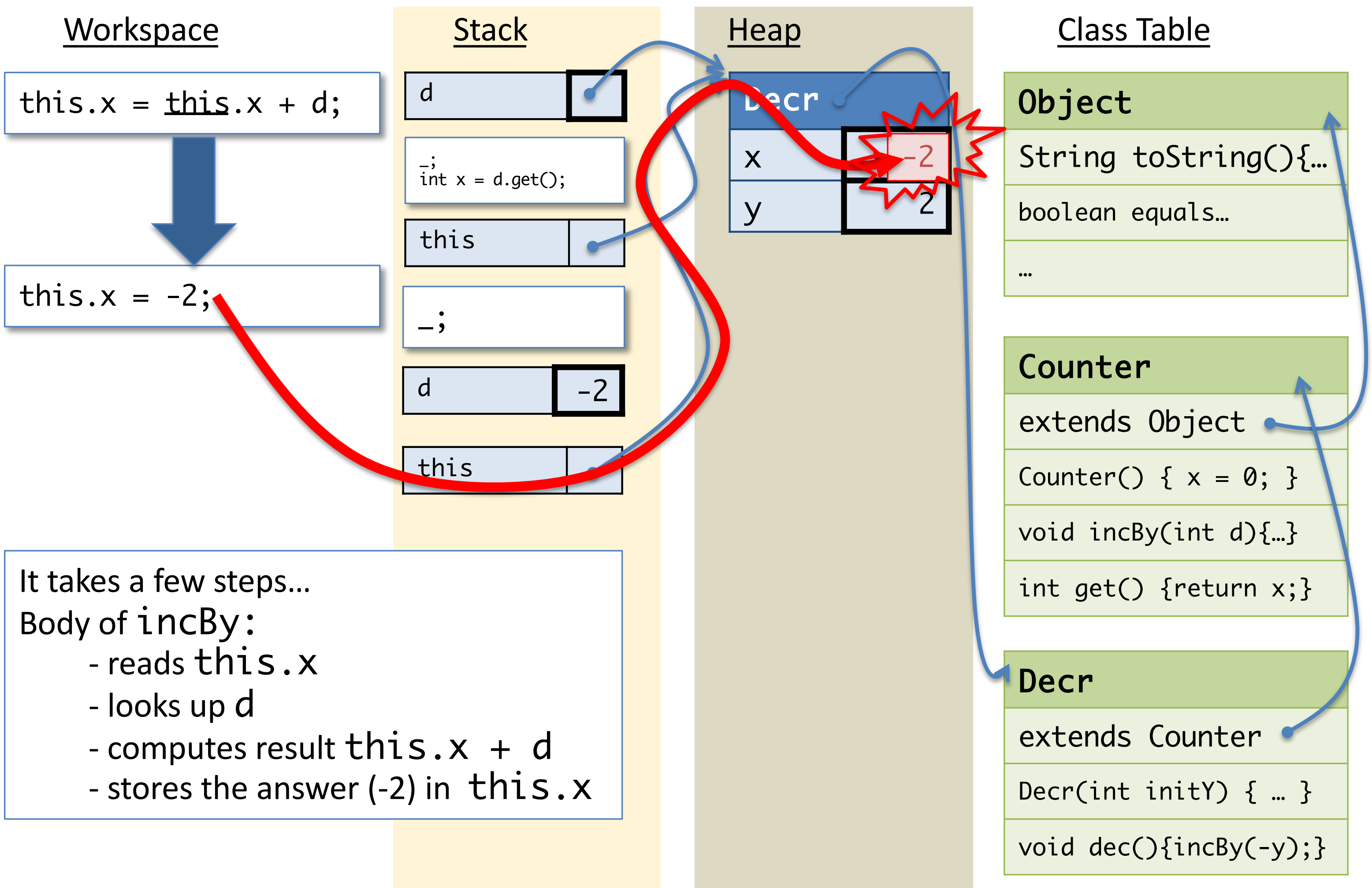
Invoke the `incBy` method on the object via dynamic dispatch.

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`

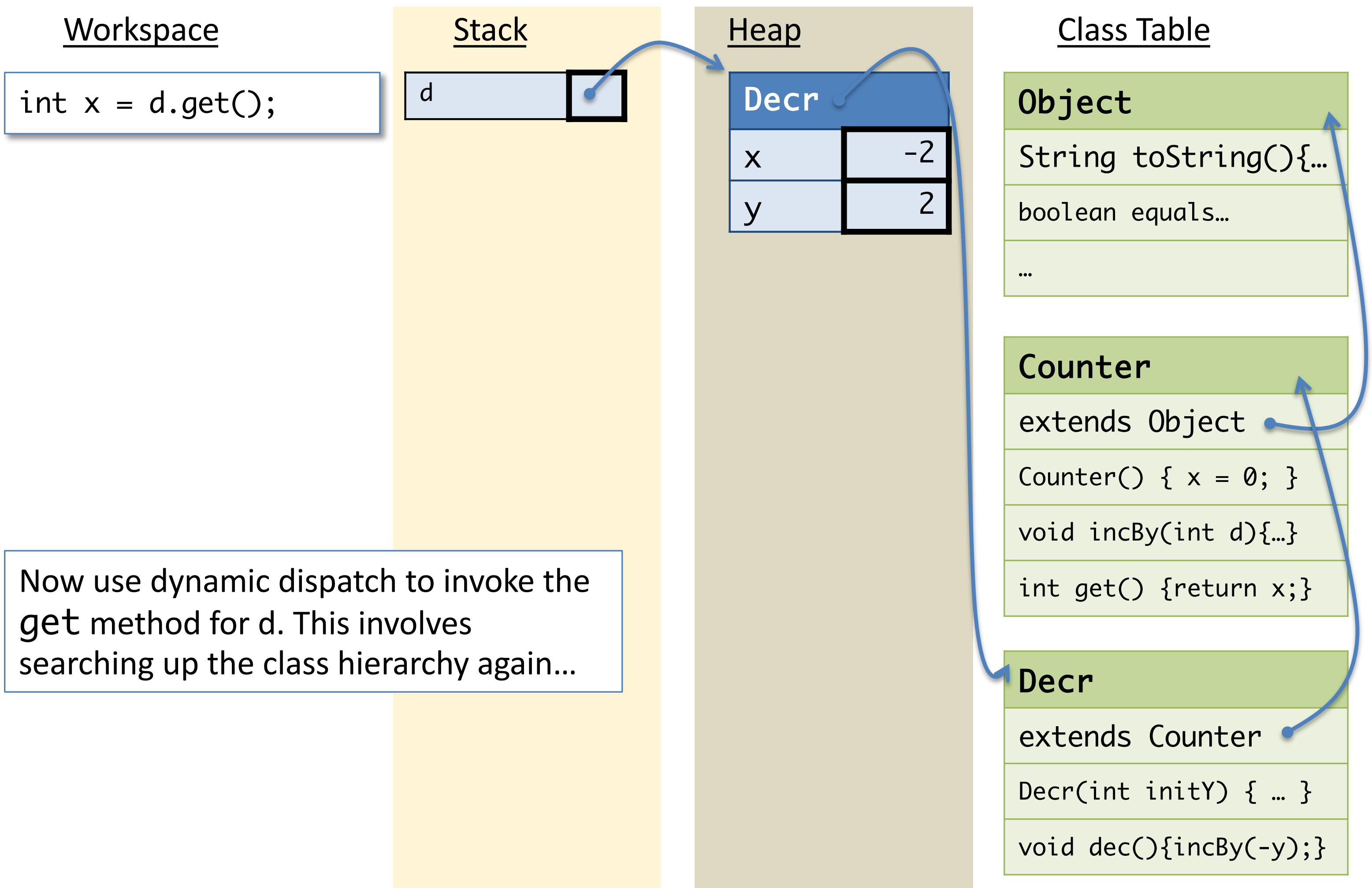


It takes a few steps…

Body of `incBy`:

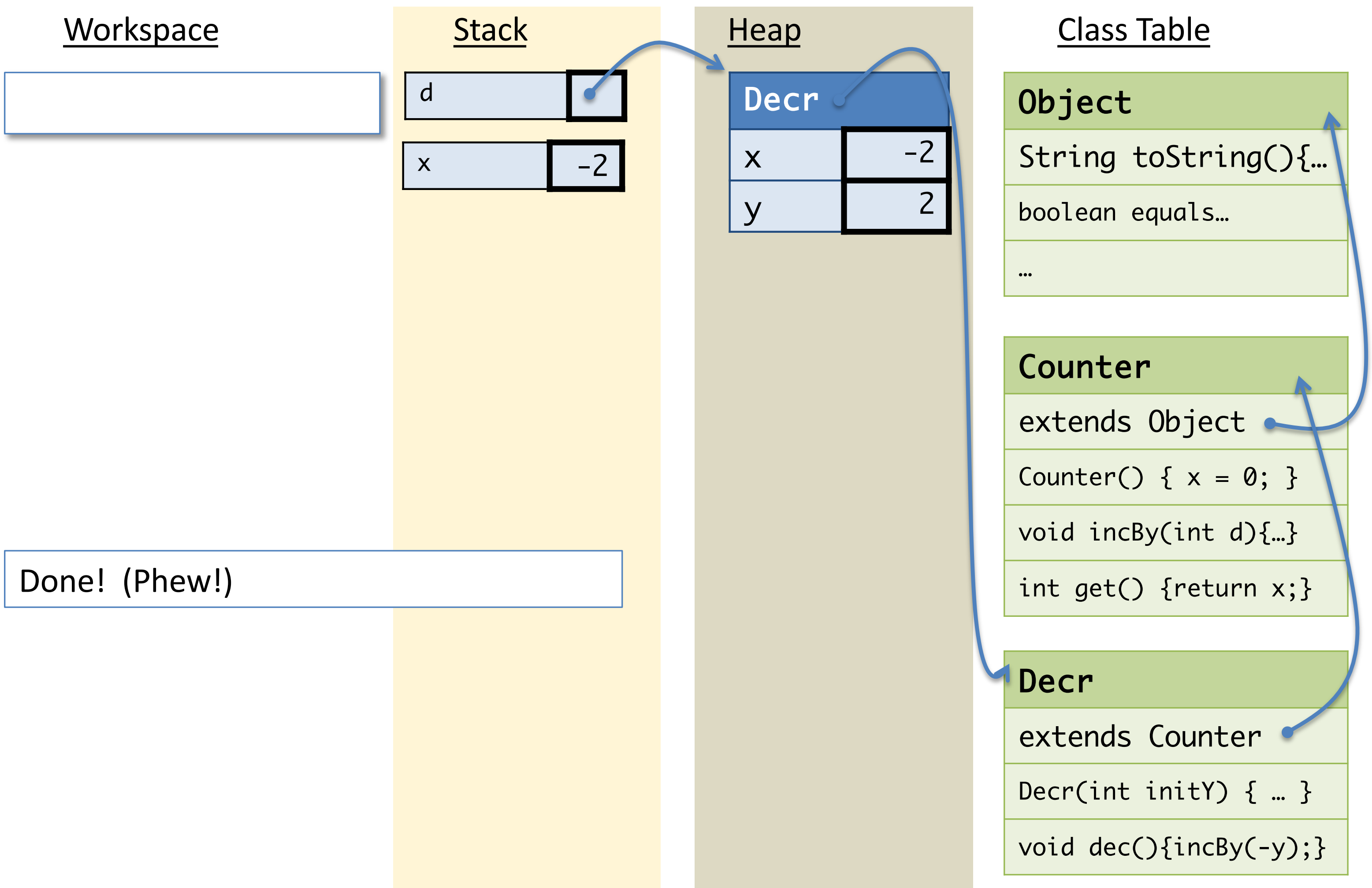
- reads `this.x`
- looks up `d`
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

After a few more steps…



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…



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.

26: What is the value of *x* at the end of this computation?

0

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

-2	0%
-1	0%
0	0%
1	0%
2	0%
NullPointerException	0%
Doesn't type check	0%

Inheritance Example

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

What is the value of x at the end of this computation?

1. -2
2. -1
3. 0
4. 1
5. 2
6. NPE
7. Doesn't type check

Answer: -2

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.

```
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);
```

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

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