

# Programming Languages and Techniques (CIS120)

## Lecture 25

Oct 30, 2015

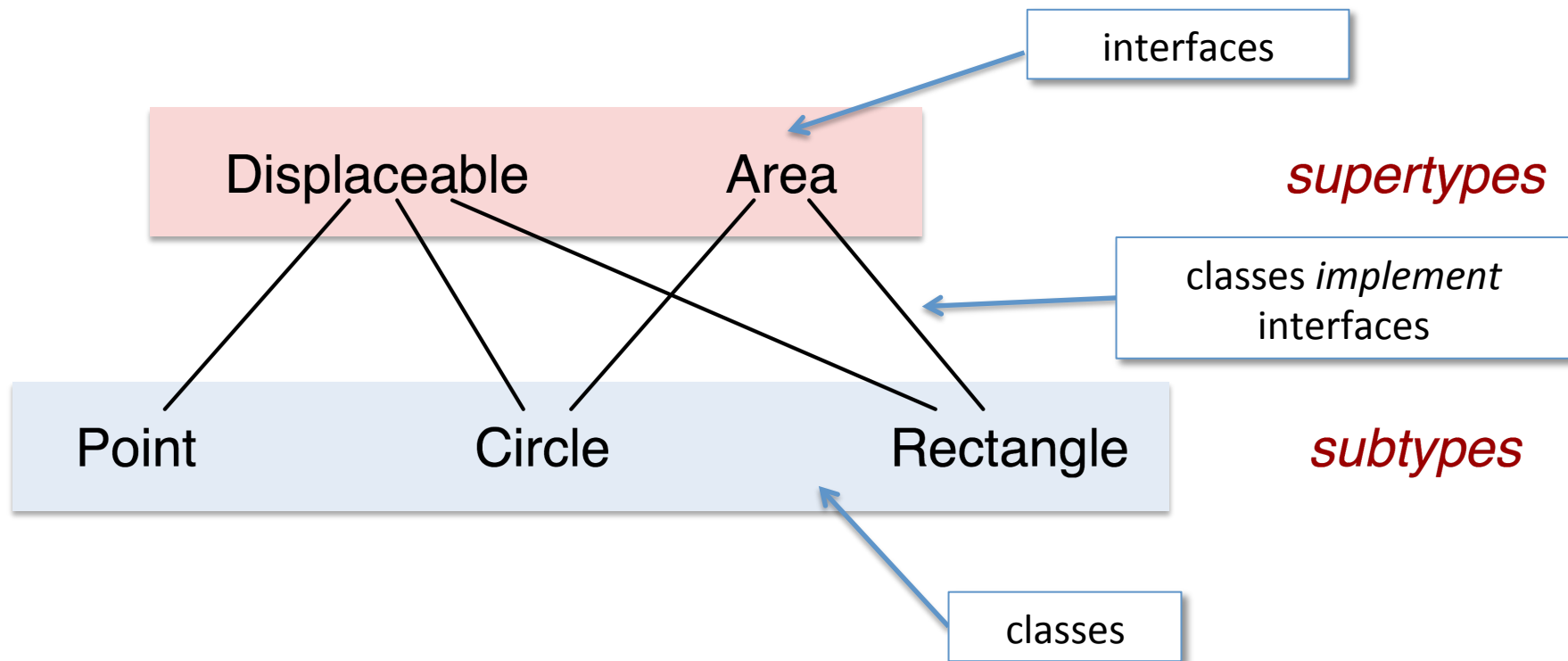
## Inheritance and Dynamic Dispatch

# Announcements

- *Midterm 2 is Friday, November 6<sup>th</sup> in class*
  - Last names A – L Leidy Labs 10 (here)
  - Last names M – Z Cohen G17
- Coverage:
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in Ocaml)
  - Arrays (in Java)
  - Subtyping & Simple Extension (in Java)
- Review Session: TBA

# 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

# Extension

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

# Interface Extension

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

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

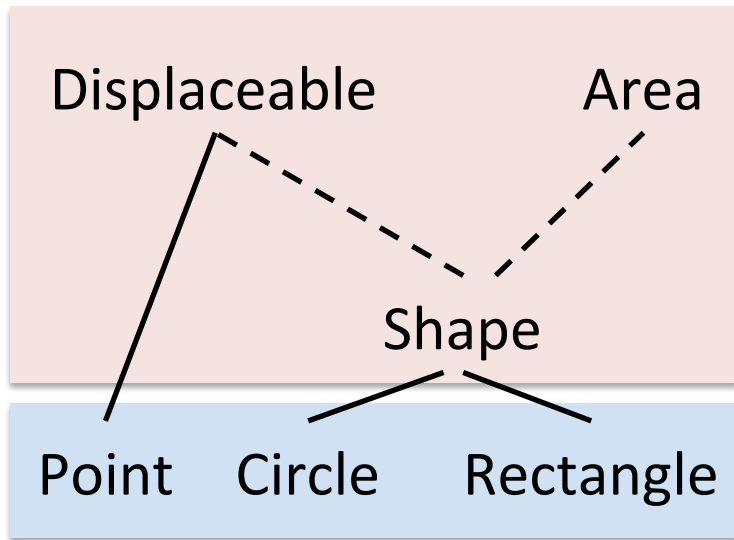
```
public interface Area {  
    double getArea();  
}
```

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

# Interface Extension Demo

See: [Shapes.zip](#)

# Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only *one* other class.
- The extending class *inherits* all of the fields and methods of its *superclass*, and may include additional fields or methods.
  - This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).
  - Class extension should *never* be used when “is a” does not relate the subtype to the supertype.

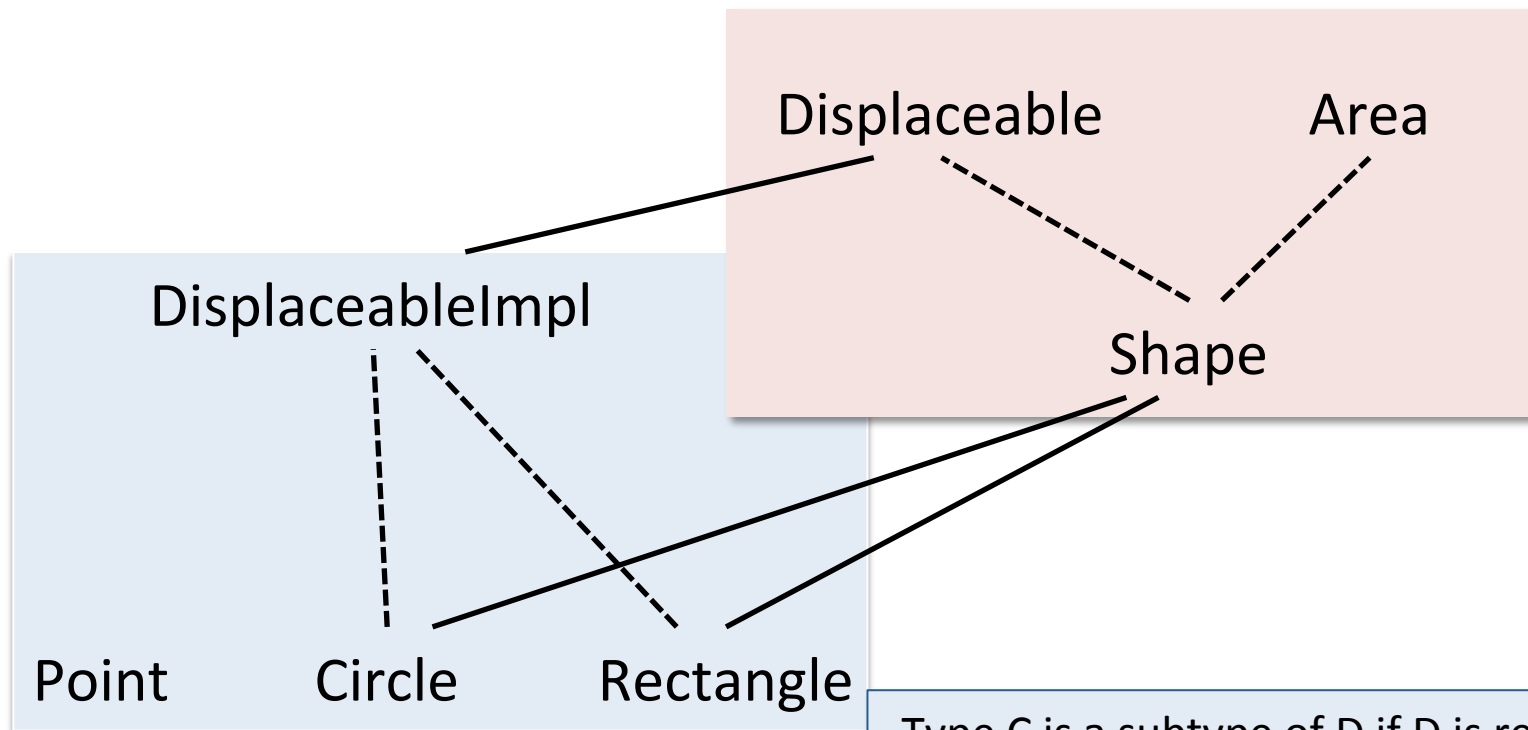
```
class D {  
    private int x;  
    private int y;  
    public int addBoth() { return x + y; }  
}  
  
class C extends D {    // every C is a D  
    private int z;  
    public int addThree() {return (addBoth() + z); }  
}
```



# Simple Inheritance

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

# 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

# Example of Simple Inheritance

See: `Main2.java`

# Inheritance: Constructors

- Constructors *cannot* be inherited (they have the wrong names!)
  - Instead, a subclass invokes the constructor of its super class using the keyword 'super'.
  - Super *must* be the first line of the subclass constructor, unless the parent class constructor takes no arguments, in which it is OK to omit the call to super (it is called implicitly).

```
class D {  
    private int x;  
    private int y;  
    public D (int initX, int initY) { x = initX; y = initY; }  
    public int addBoth() { return x + y; }  
}
```

```
class C extends D {  
    private int z;  
    public C (int initX, int initY, int initZ) {  
        super(initX, initY);  
        z = initZ;  
    }  
    public int addThree() {return (addBoth() + z); }  
}
```

# 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 hard to use properly, and the need for them arises only in somewhat special cases
  - Making reusable libraries
  - Special methods: equals and toString
- We recommend avoiding *all* forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition.

*Especially: avoid overriding.*

When do constructors execute?  
How are fields accessed?  
What code runs in a method call?

# Revenge of the Son of the Abstract Stack Machine

# How do method calls work?

- What code gets run in a method invocation?

`o.move(3,4);`

- When that code is running, how does it access the fields of the object that invoked it?

`x = x + dx;`

- When does the code in a constructor get executed?
- What if the method was inherited from a superclass?



# 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

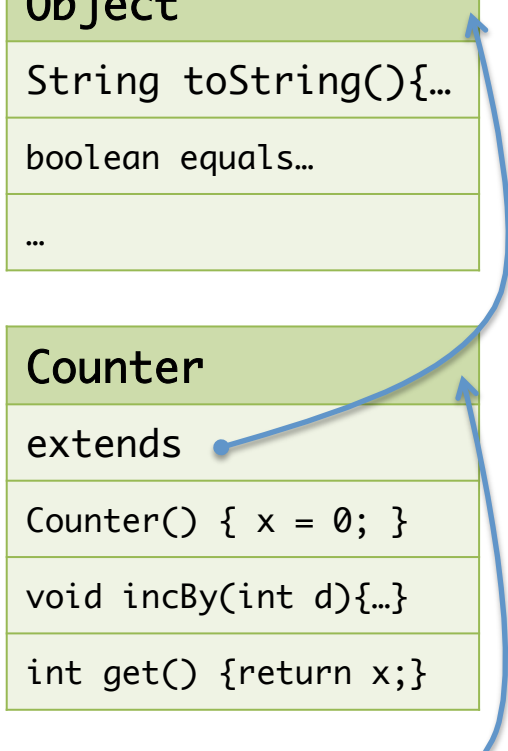
Object
String toString(){...}
boolean equals...
...

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

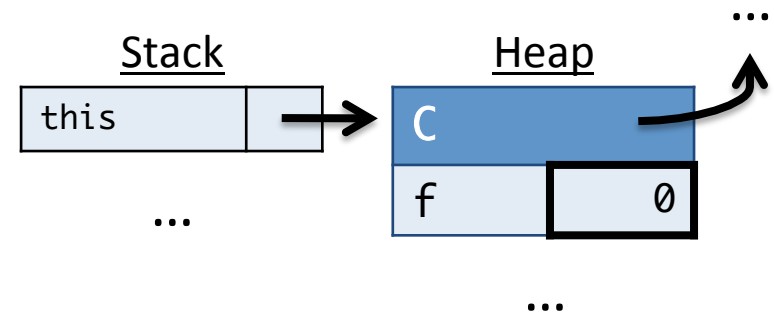
Decr
extends 
Decr(int initY) { ... }
void dec(){incBy(-y);}



# this

- Inside a non-static method, the variable `this` is a 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){...}

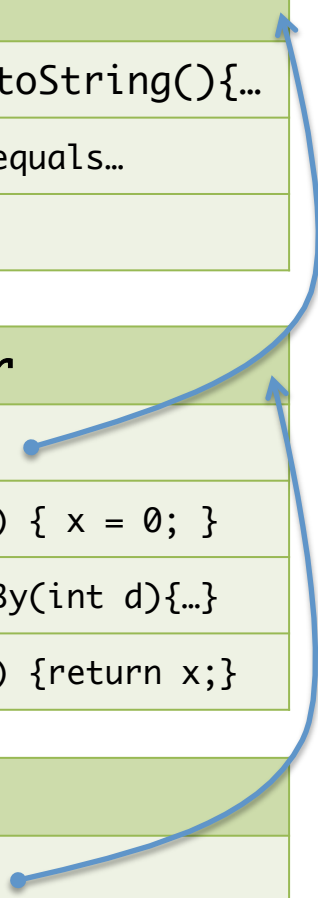
int get() {return x;}

**Decr**

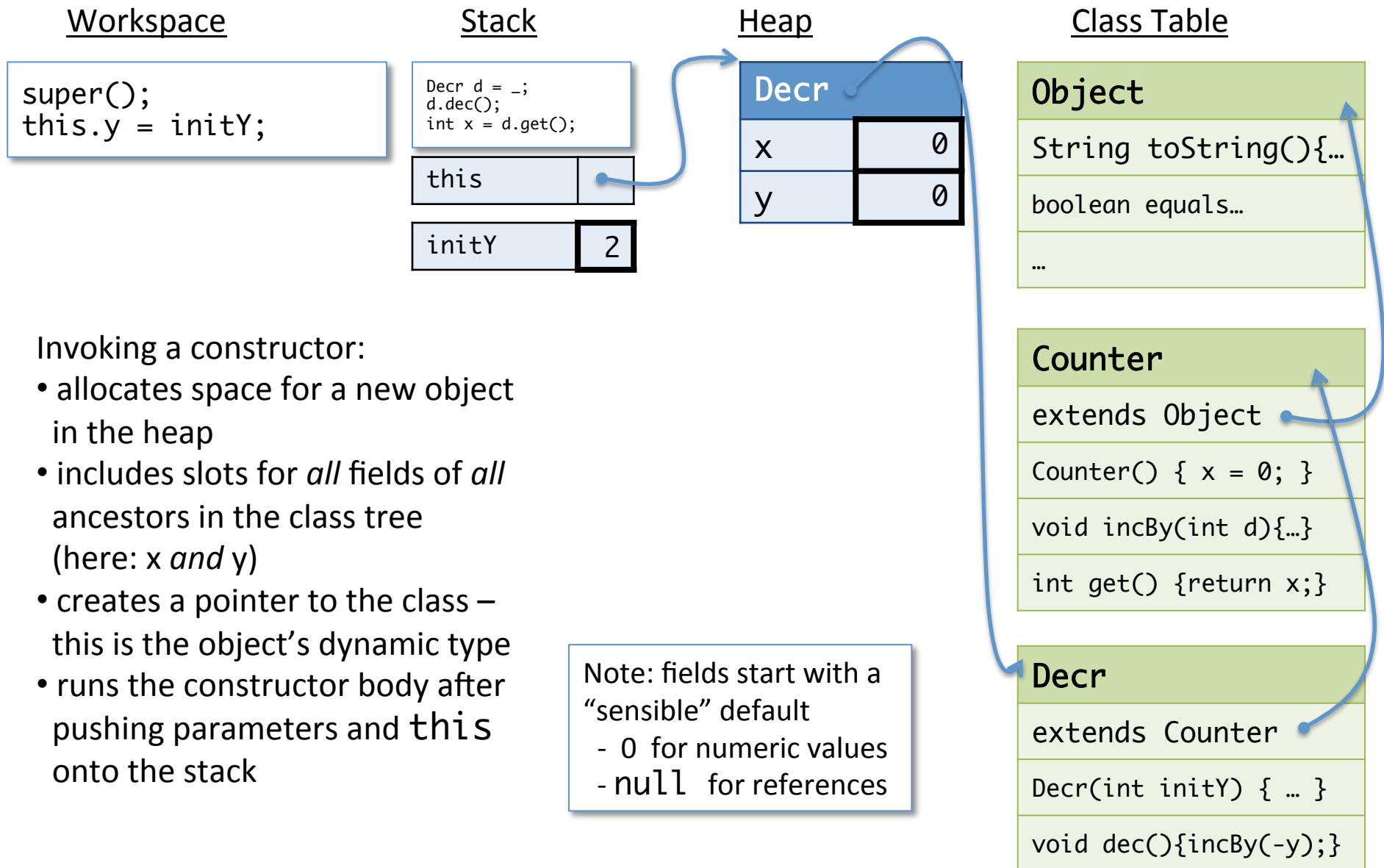
extends

Decr(int initY) { ... }

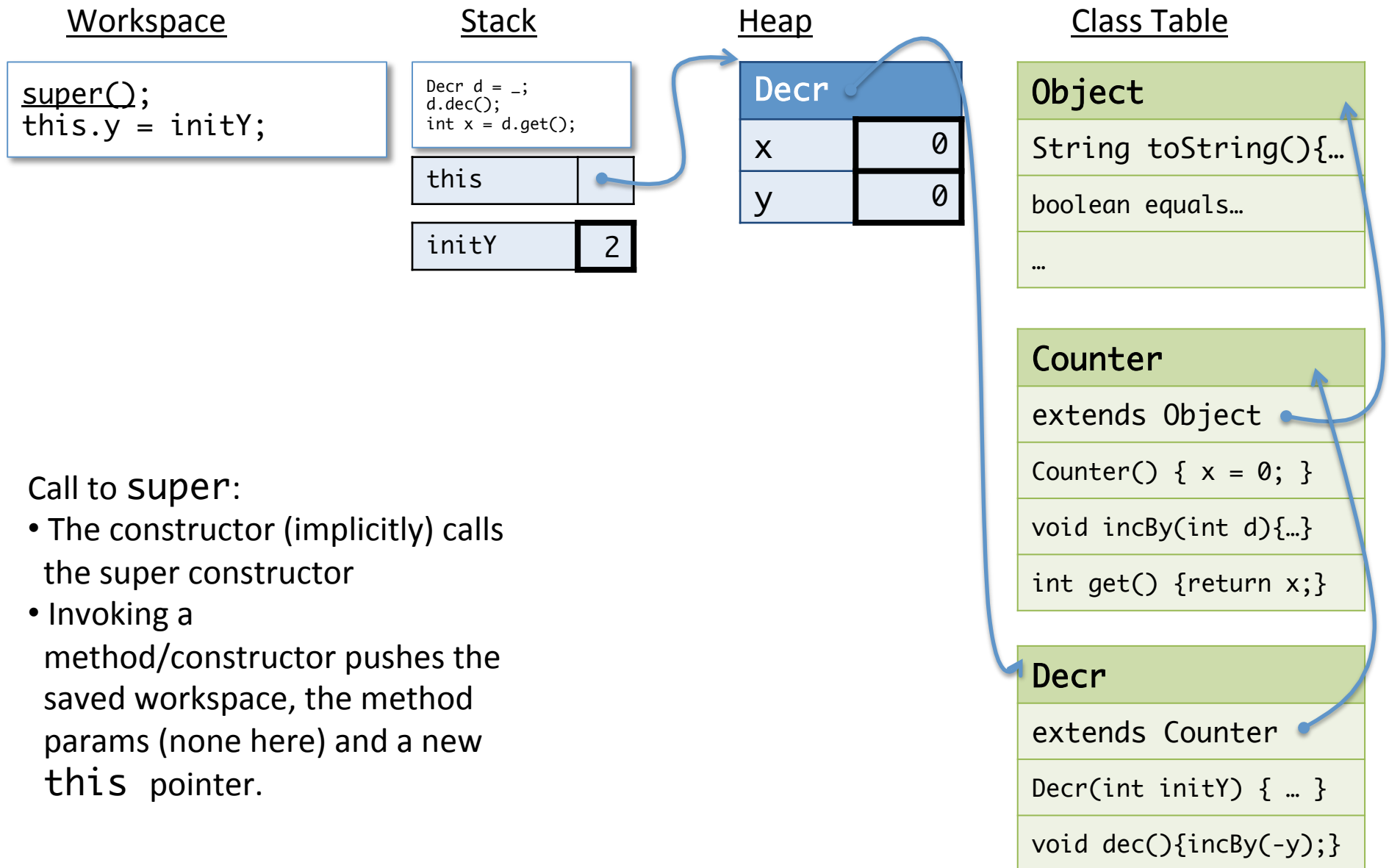
void dec(){incBy(-y);}



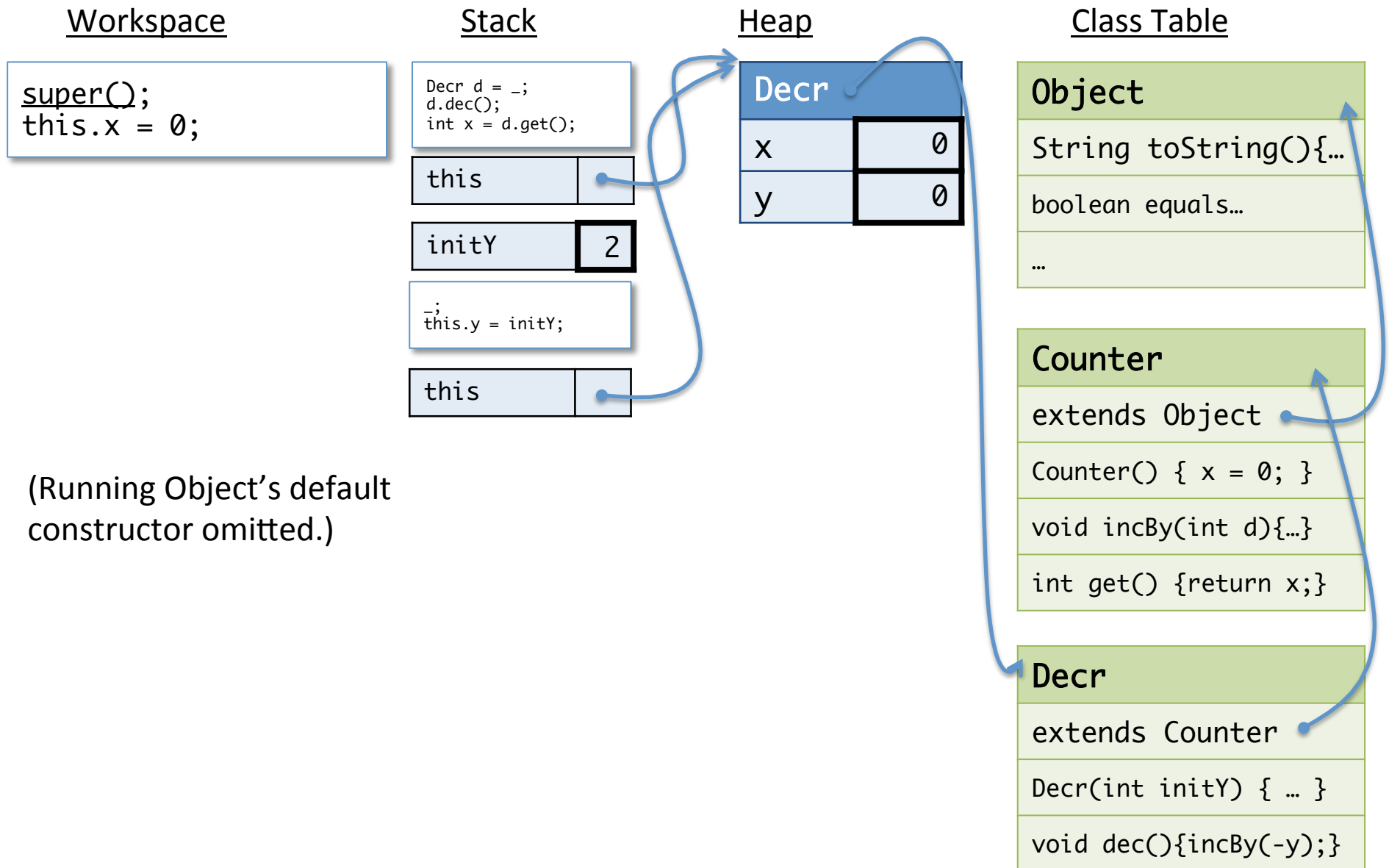
# Allocating Space on the Heap



# Calling super

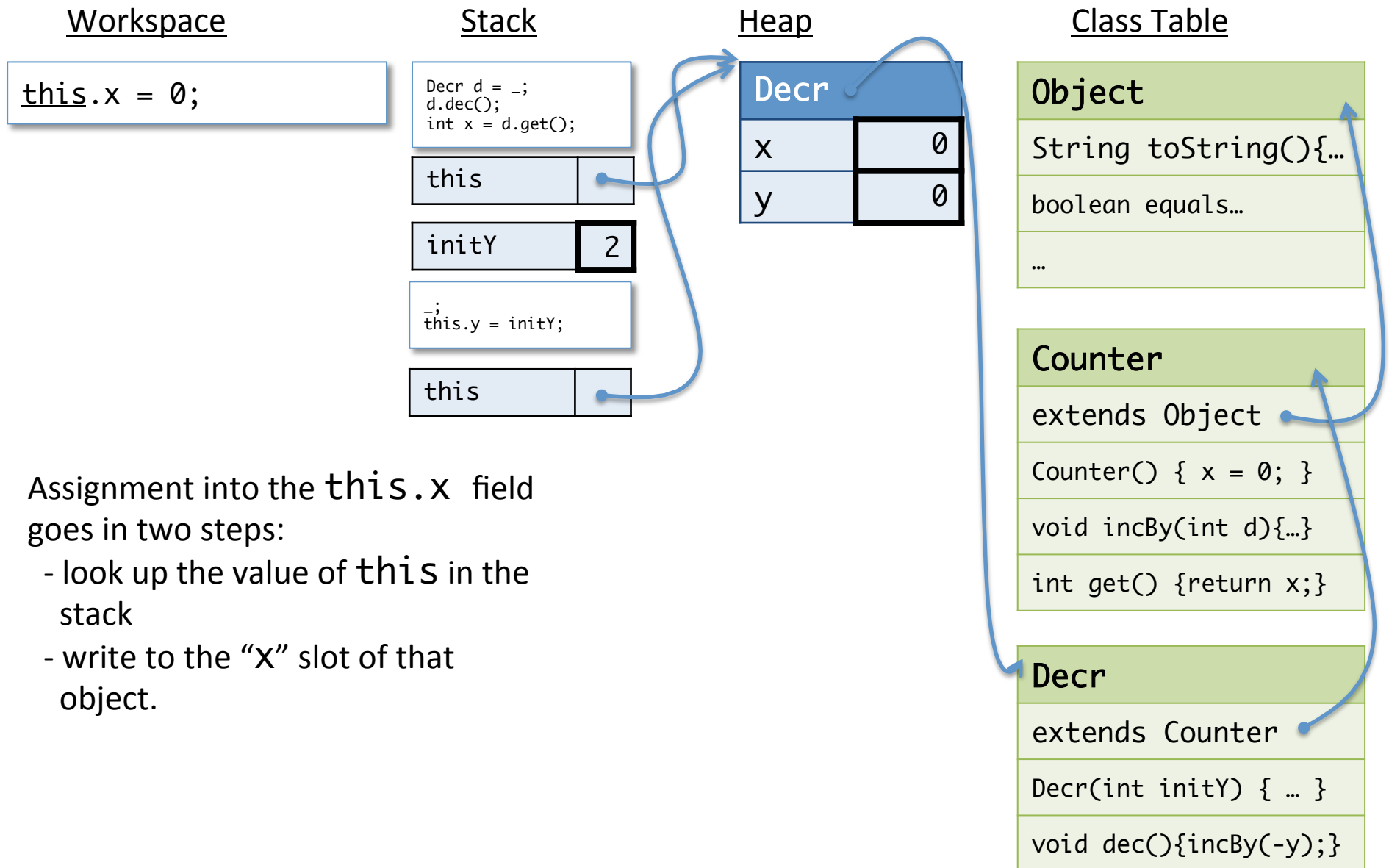


# Abstract Stack Machine





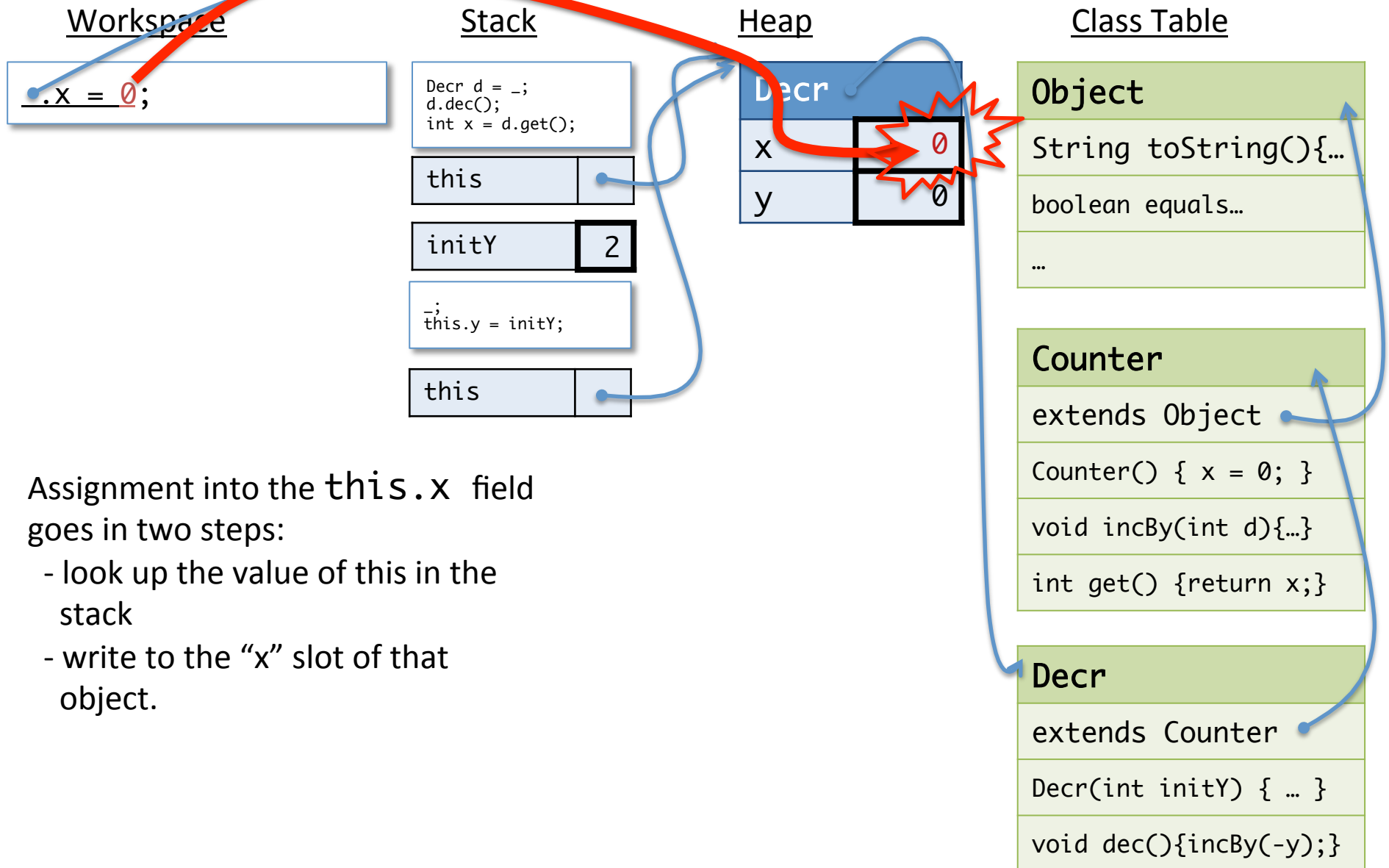
# Assigning to a Field



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.

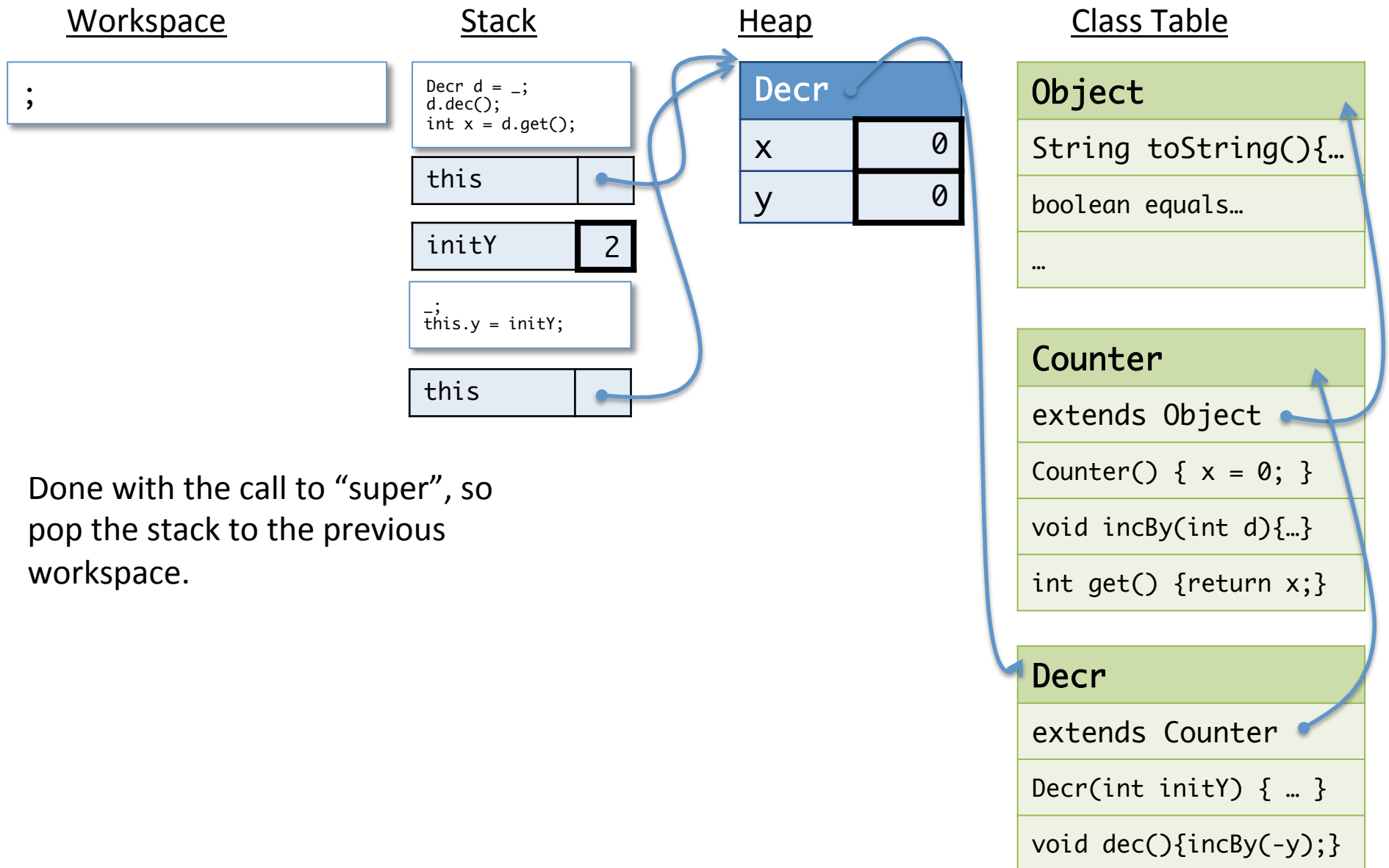
# Assigning to a Field



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



# 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**

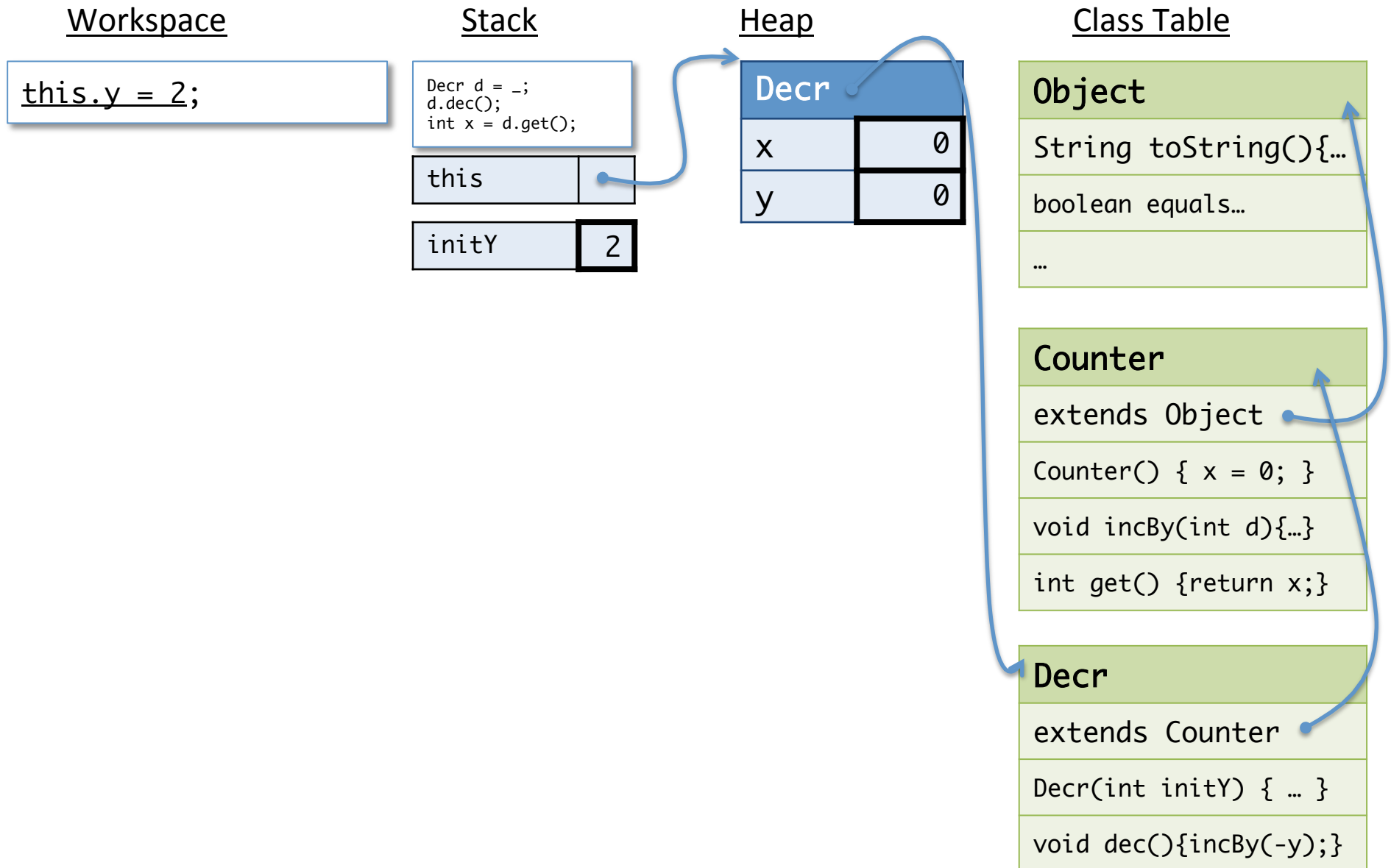
extends Counter

Decr(int initY) { ... }

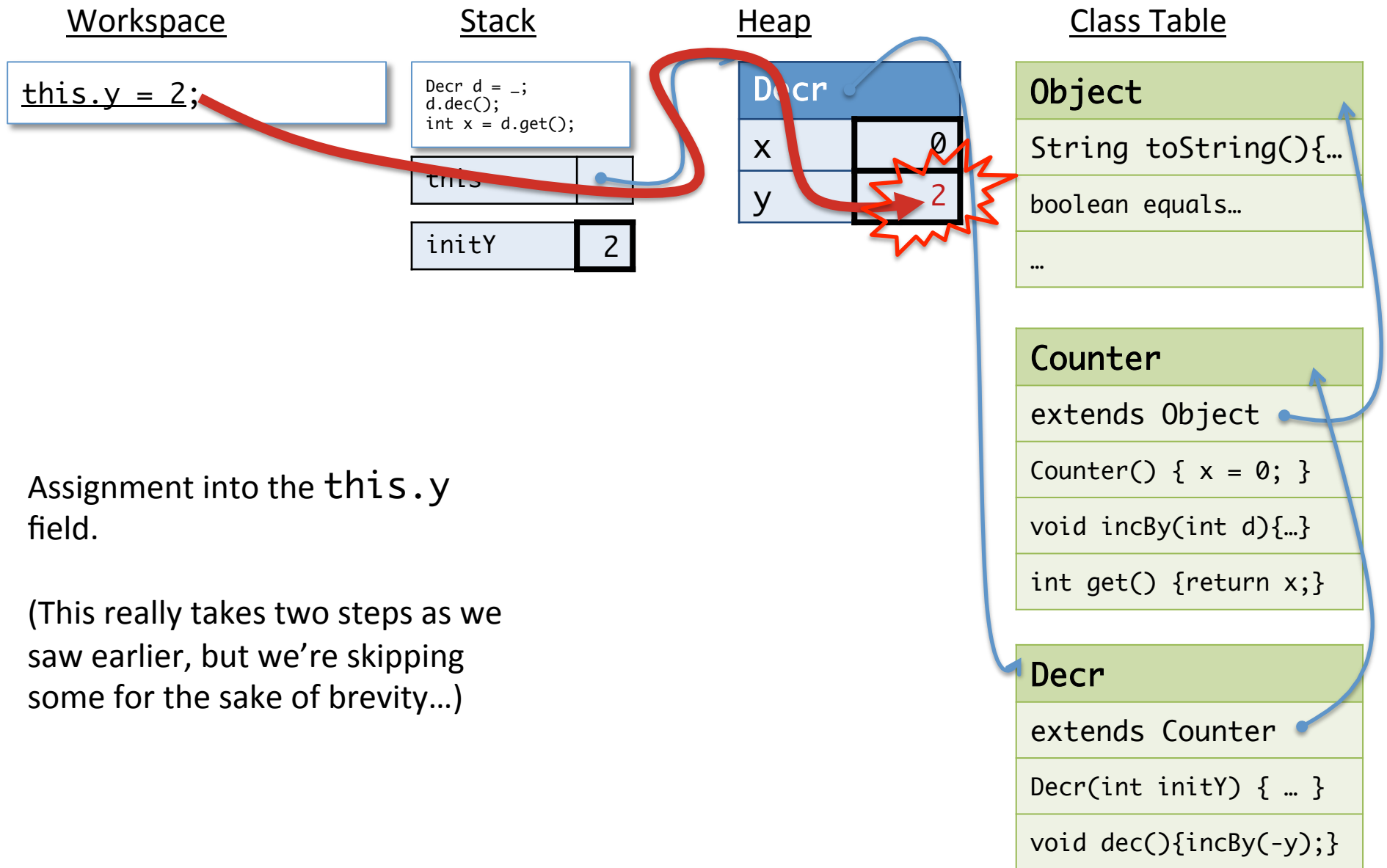
void dec(){incBy(-y);}

Continue in the Decr class's constructor.

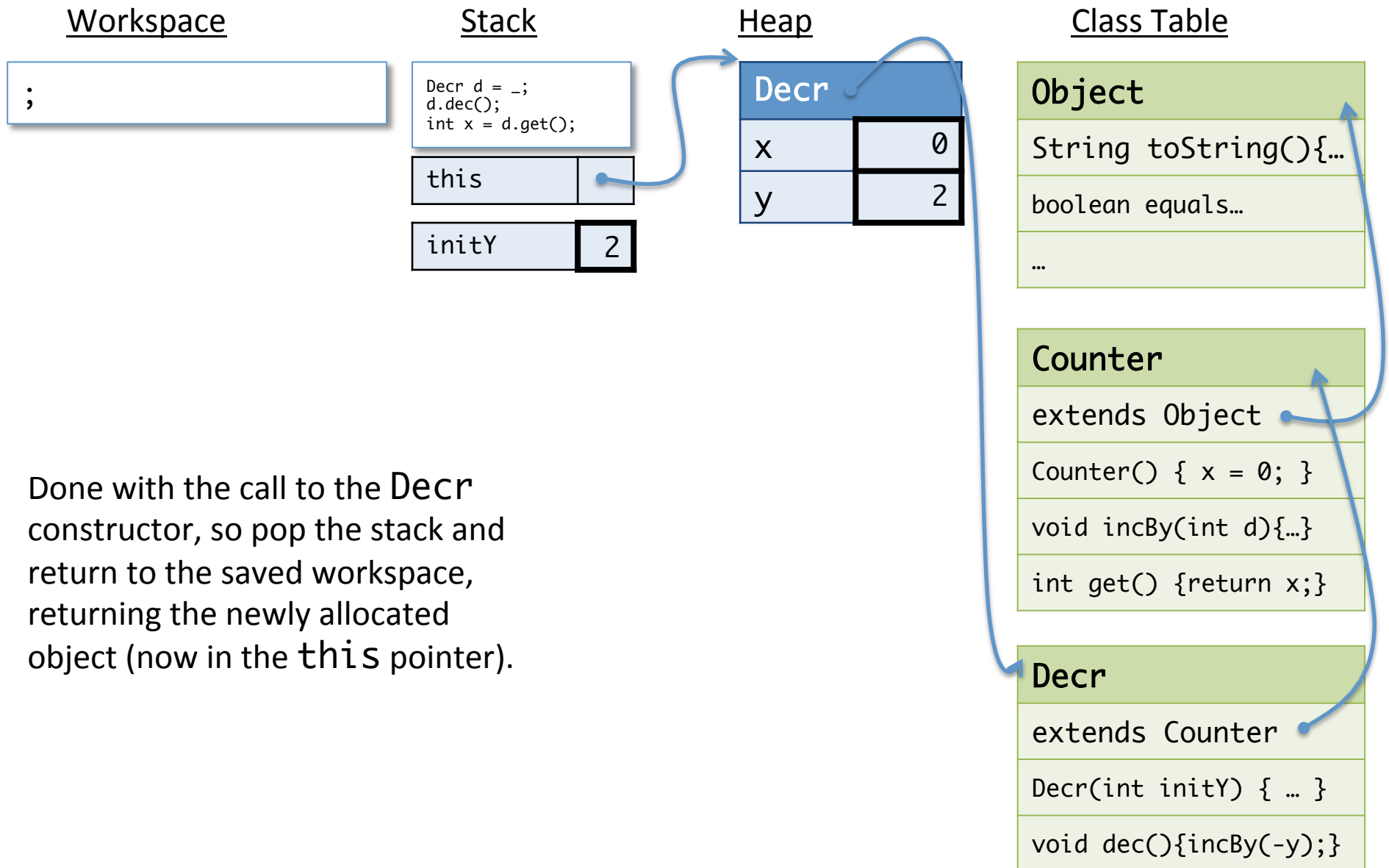
# Abstract Stack Machine



# Assigning to a field

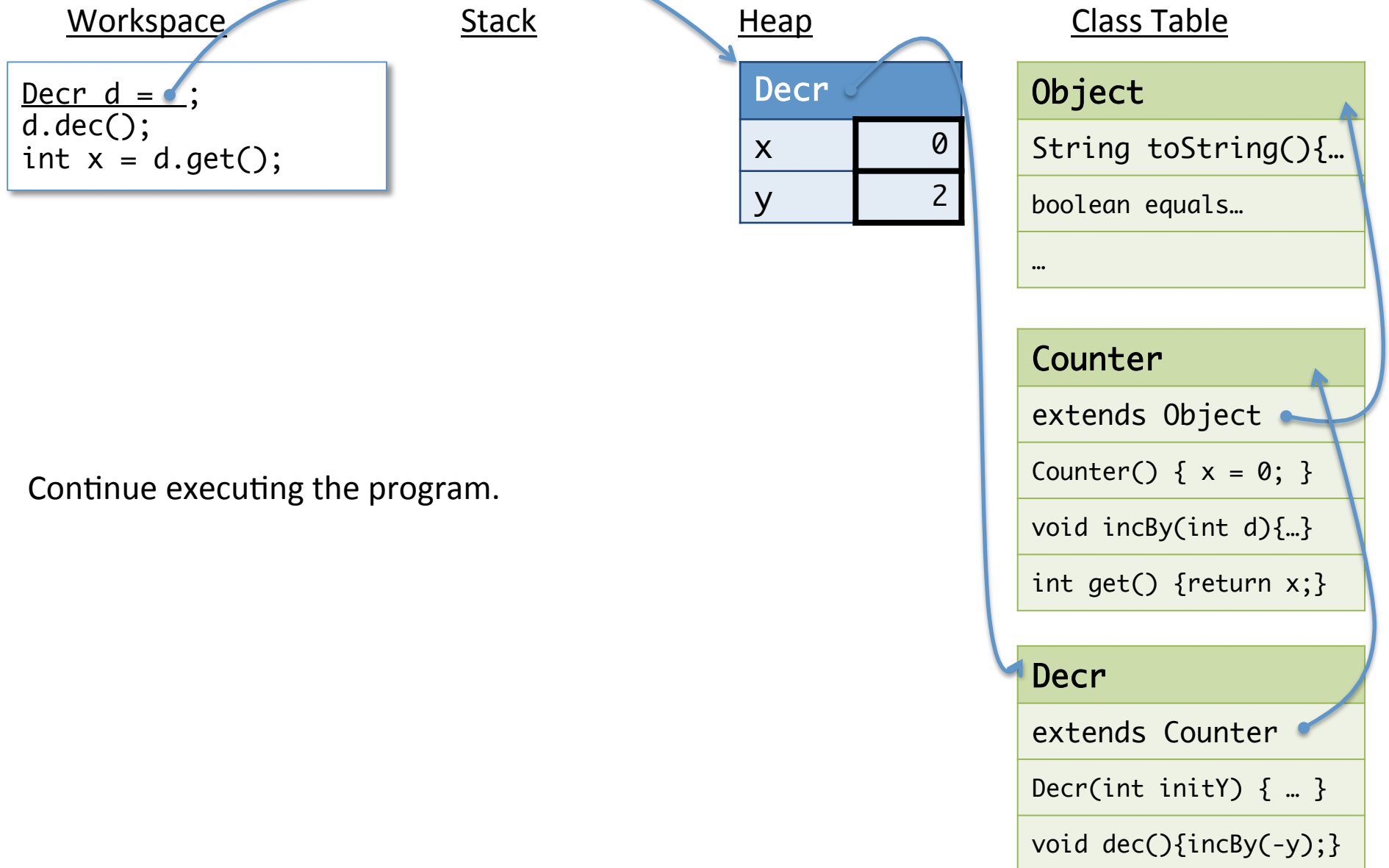


# Done with the call



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).

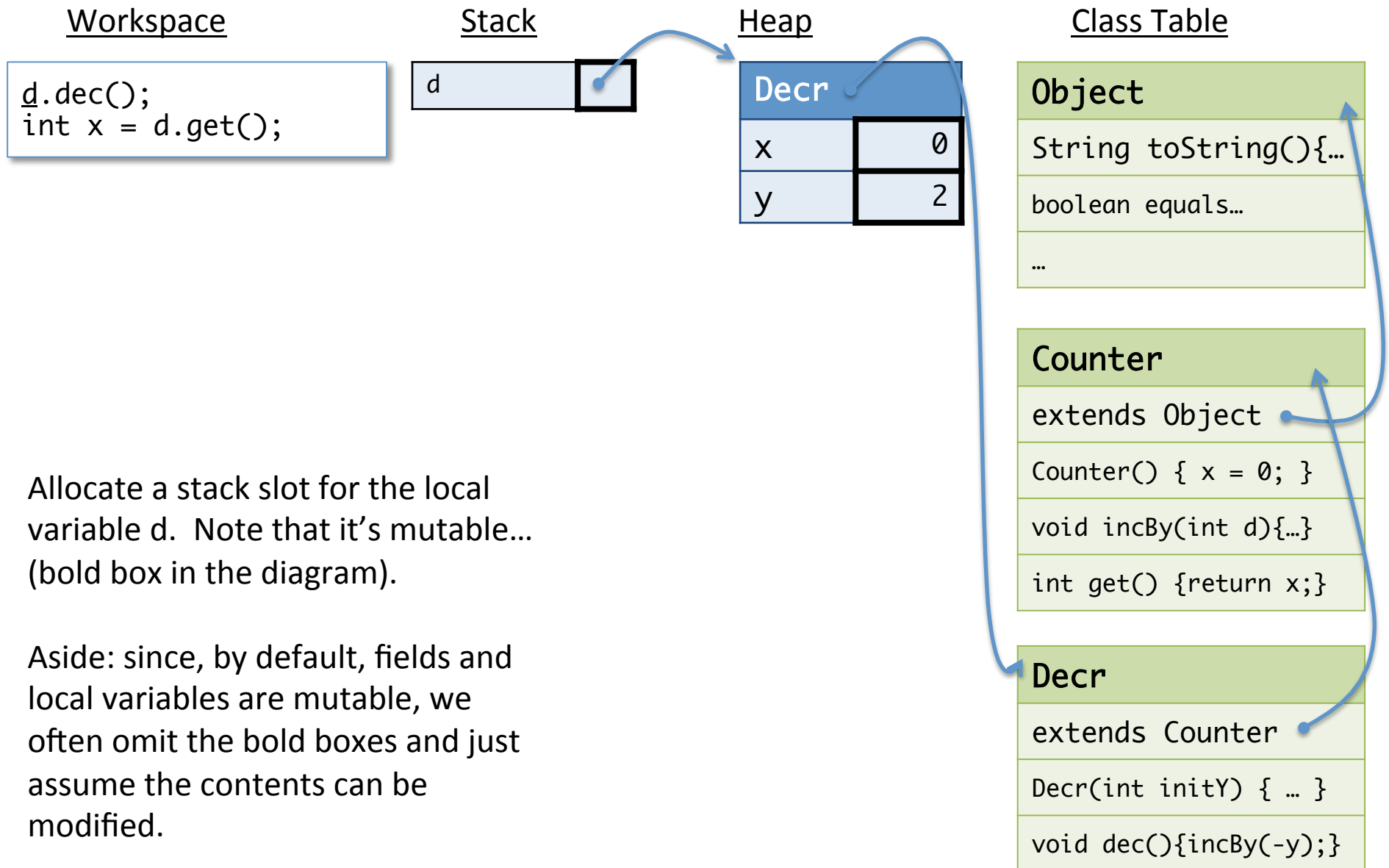
# Returning the Newly Constructed Object



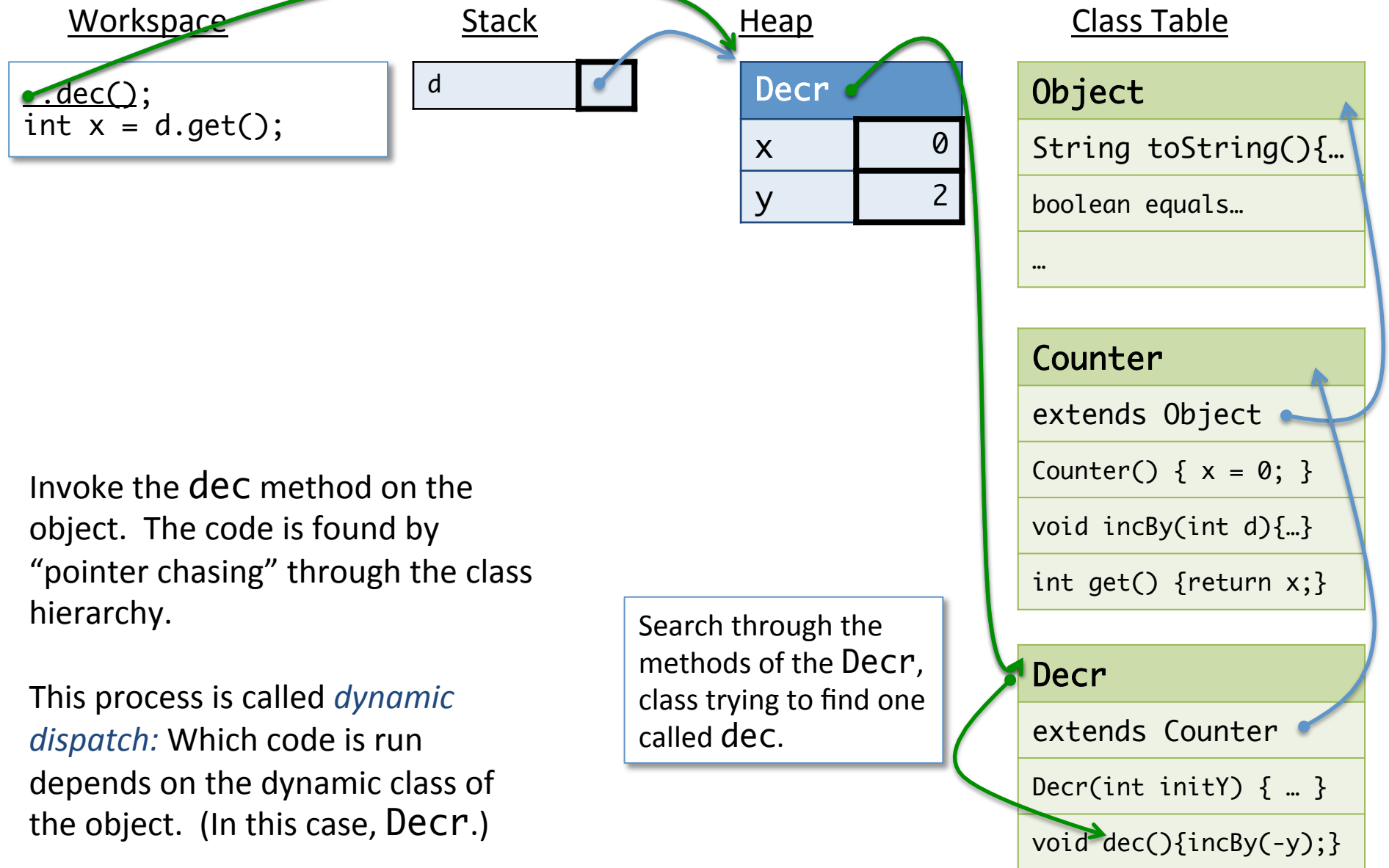
Continue executing the program.



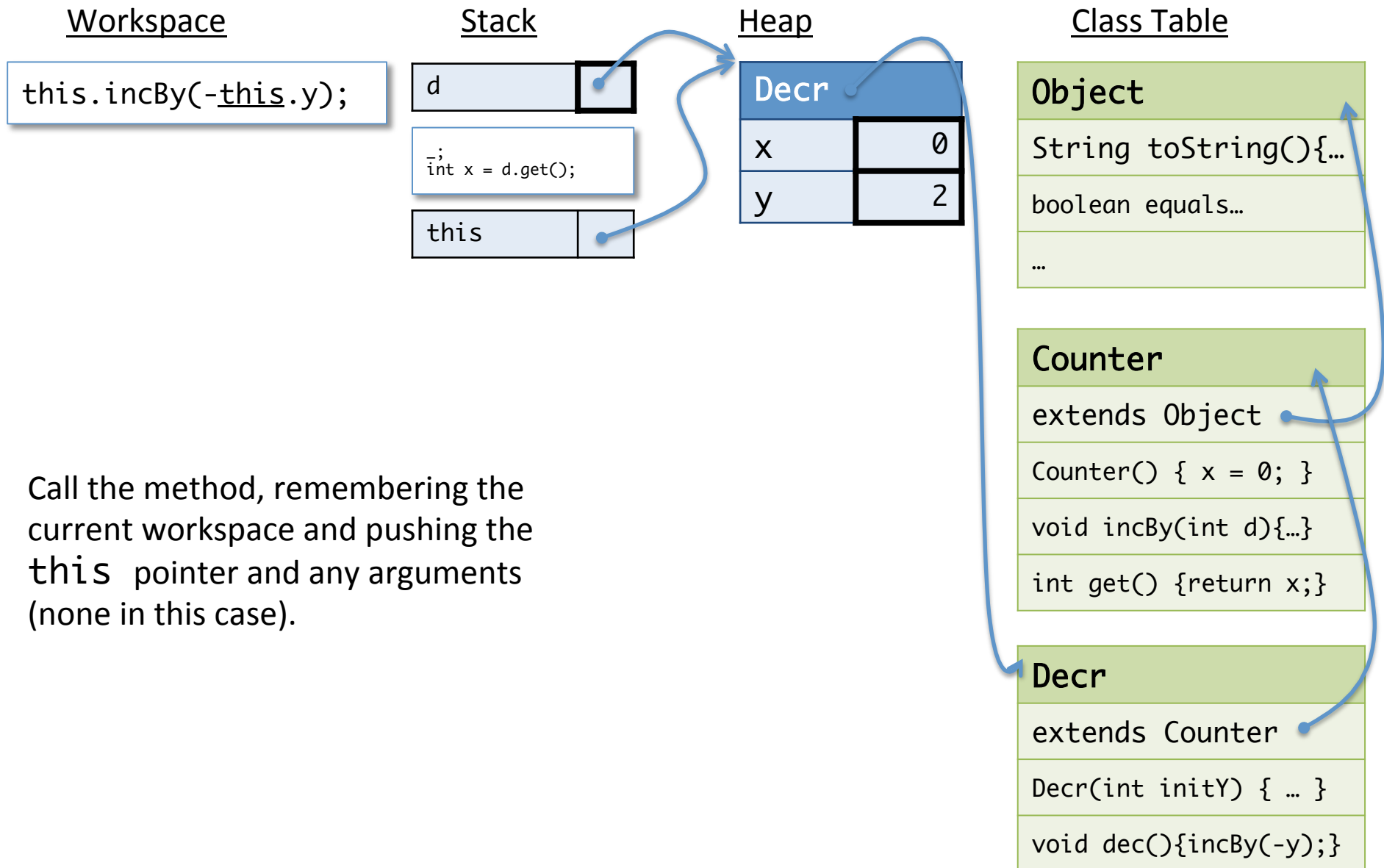
# Allocating a local variable



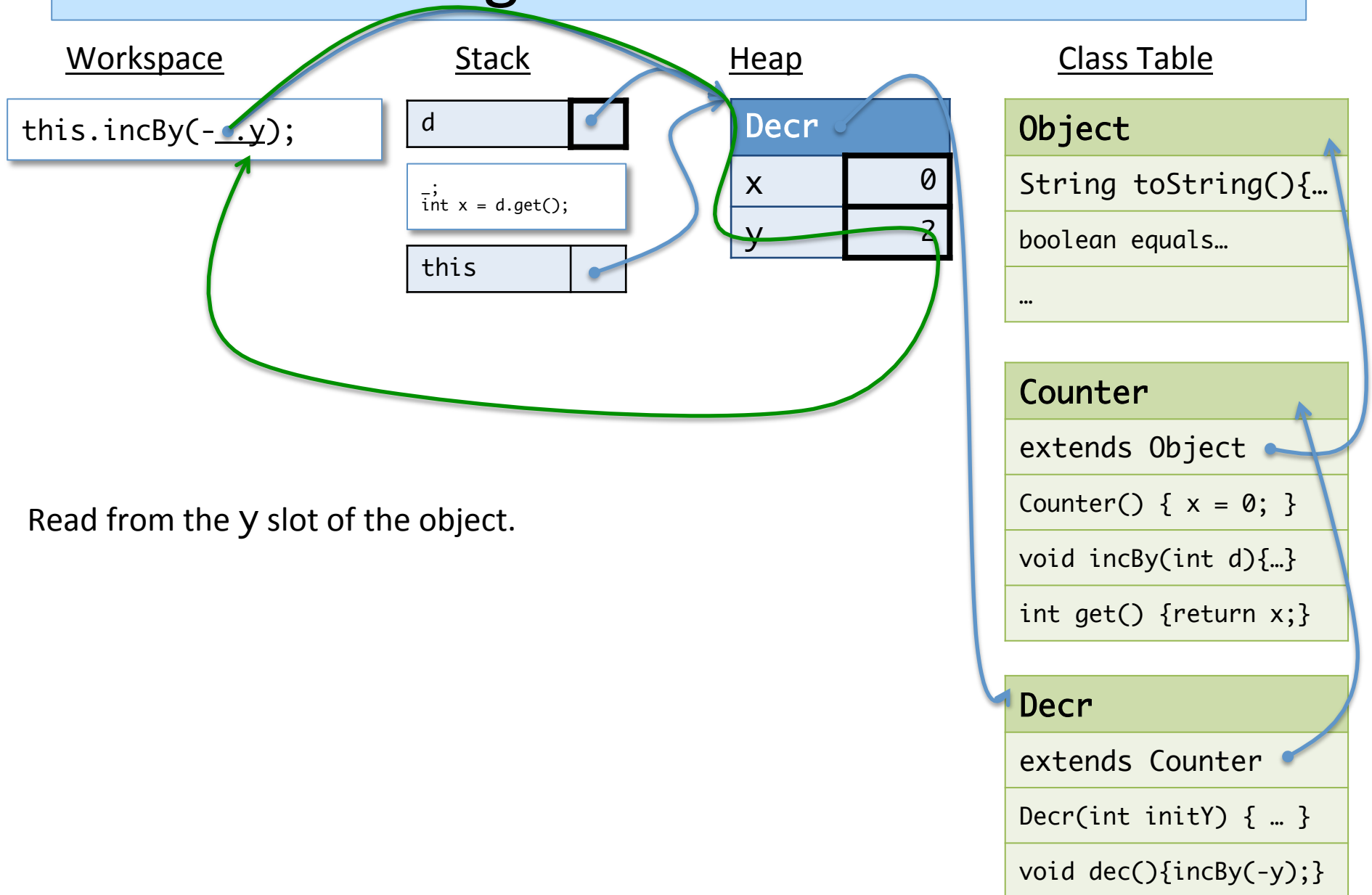
# Dynamic Dispatch: Finding the Code



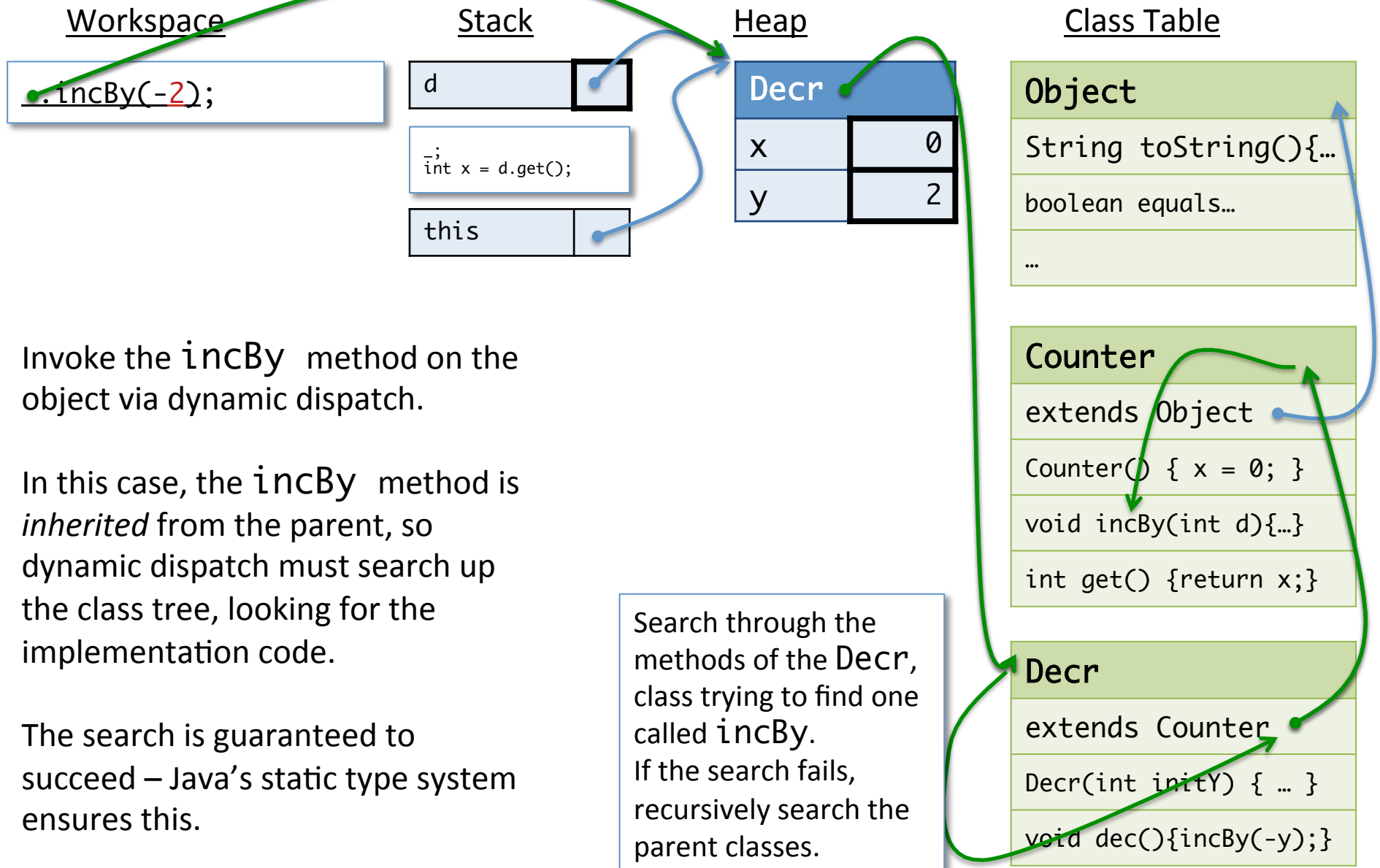
# Dynamic Dispatch: Finding the Code



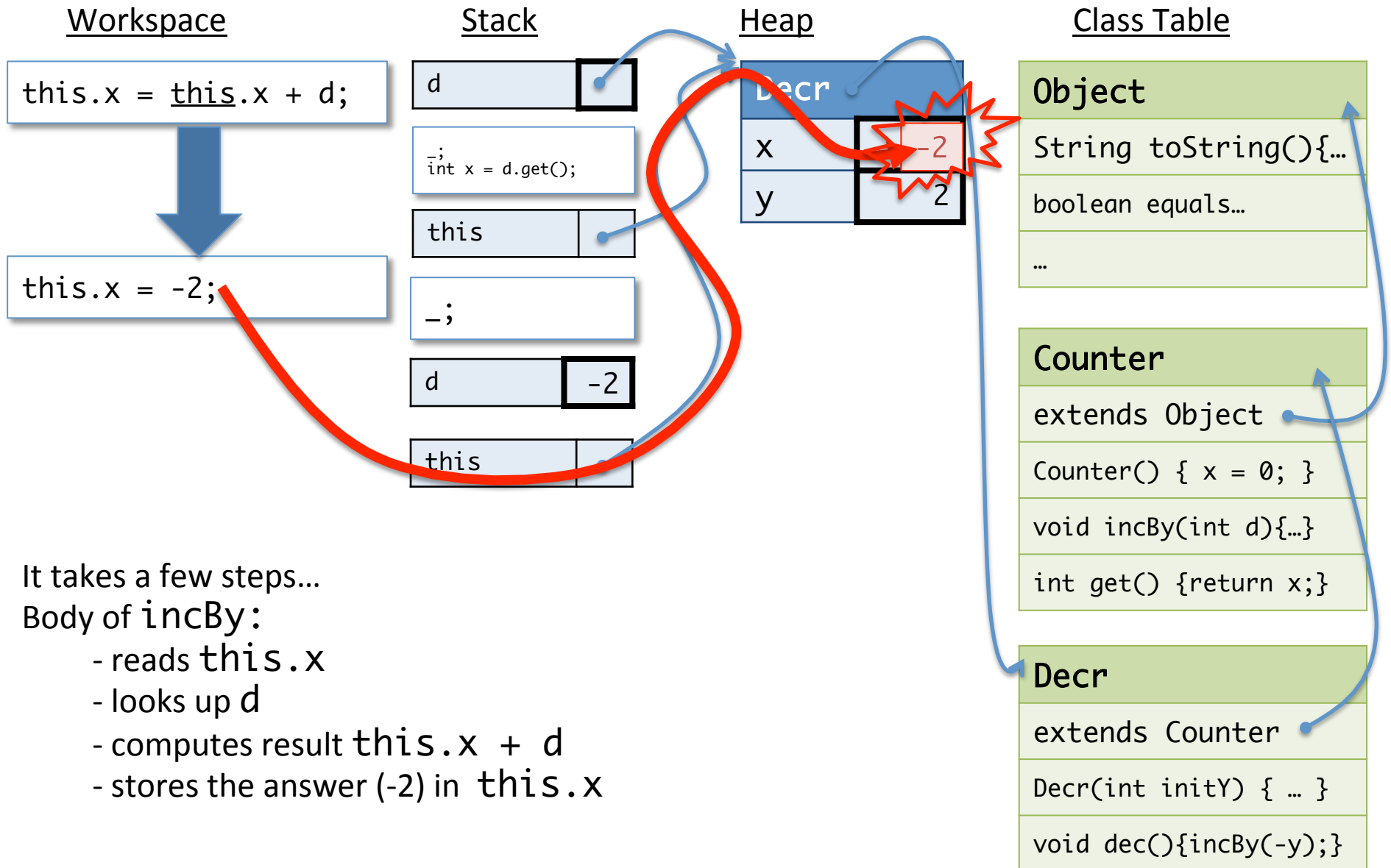
# Reading A Field's Contents



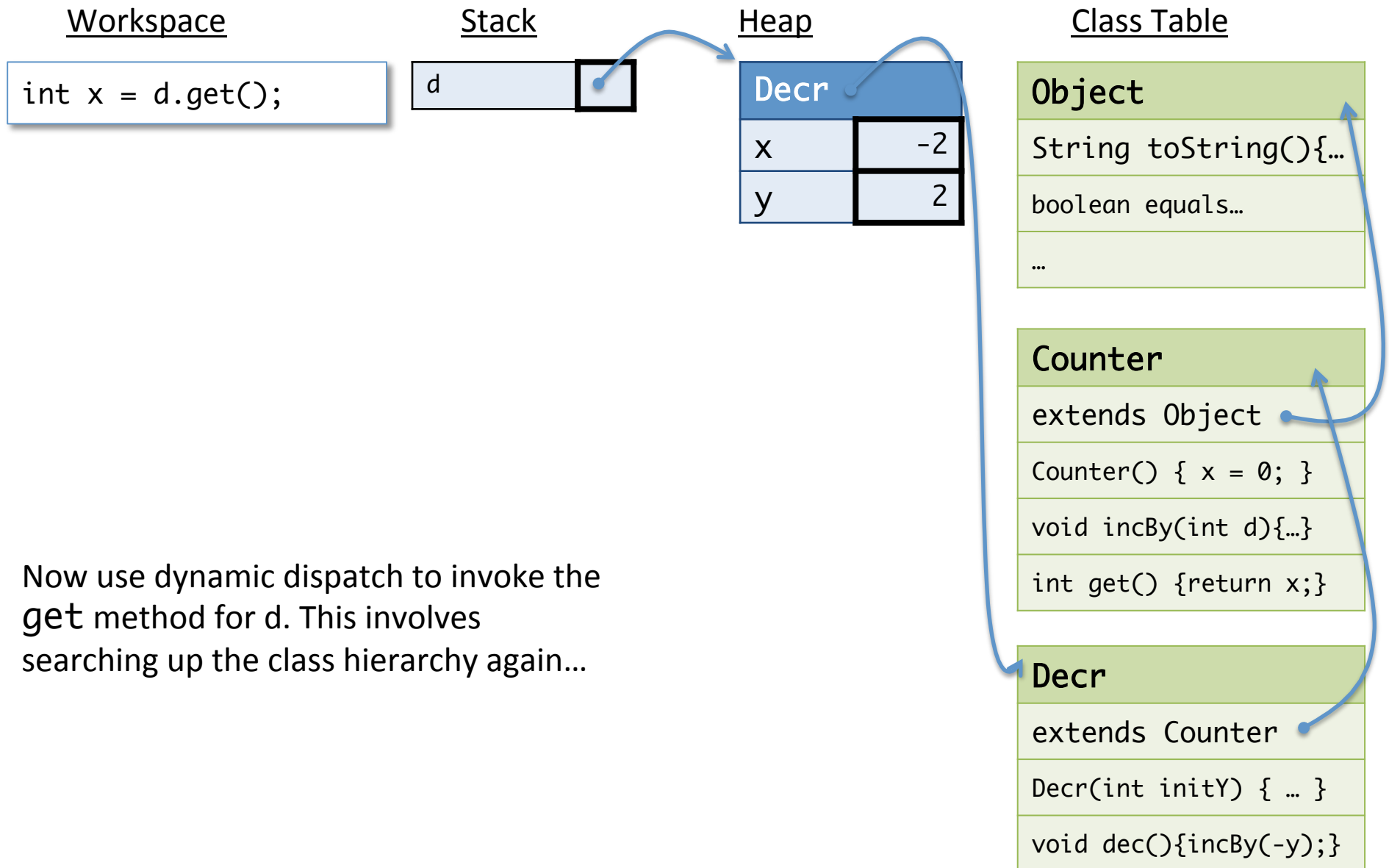
# Dynamic Dispatch, Again



# Running the body of incBy

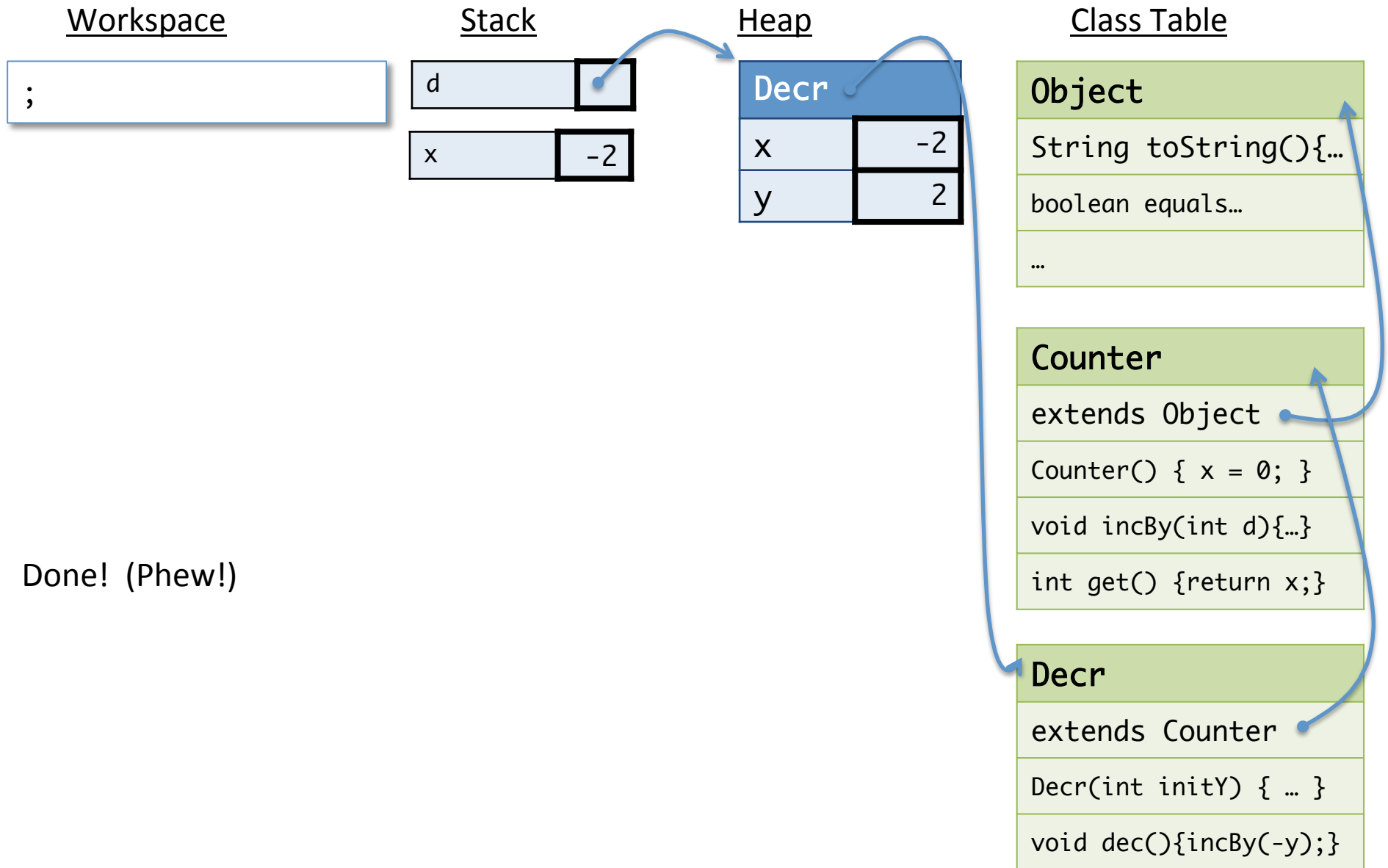


# After a few more steps...



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.