Programming Languages and Techniques (CIS1200)

Lecture 26

The Java ASM, Dynamic Dispatch

Chapter 24

Announcements (1)

- HW07: PennPals
  - Programming with Java Collections
  - Available soon
  - Due Tuesday, November 12 at 11.59pm

- Midterm 2: Friday, November 15th
  - Similar to Midterm 1
  - Content: HW 4 – 6, Chapters 11-21 (Java Arrays) and Chapter 32 (Encapsulation) of lecture notes

Announcements (2)

- Midterm 2: Friday, November 15
  - Coverage: up to Monday, Oct. 28 (Chapters 11-21, 32)
  - During lecture (001 @ 10.15am, 002 @ noon)
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site ("schedule" tab)
  - Review Session
    - TBA

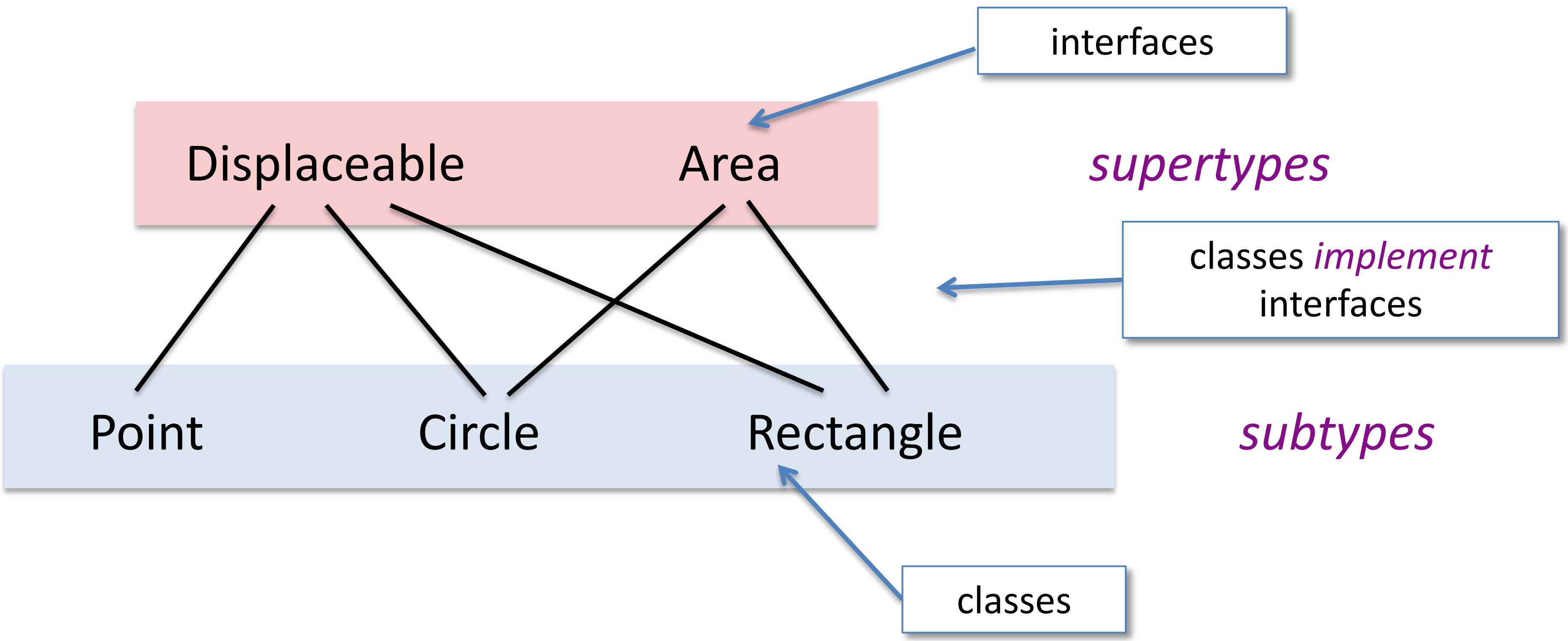
Subtyping

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 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*

- 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 nonstatic 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

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.

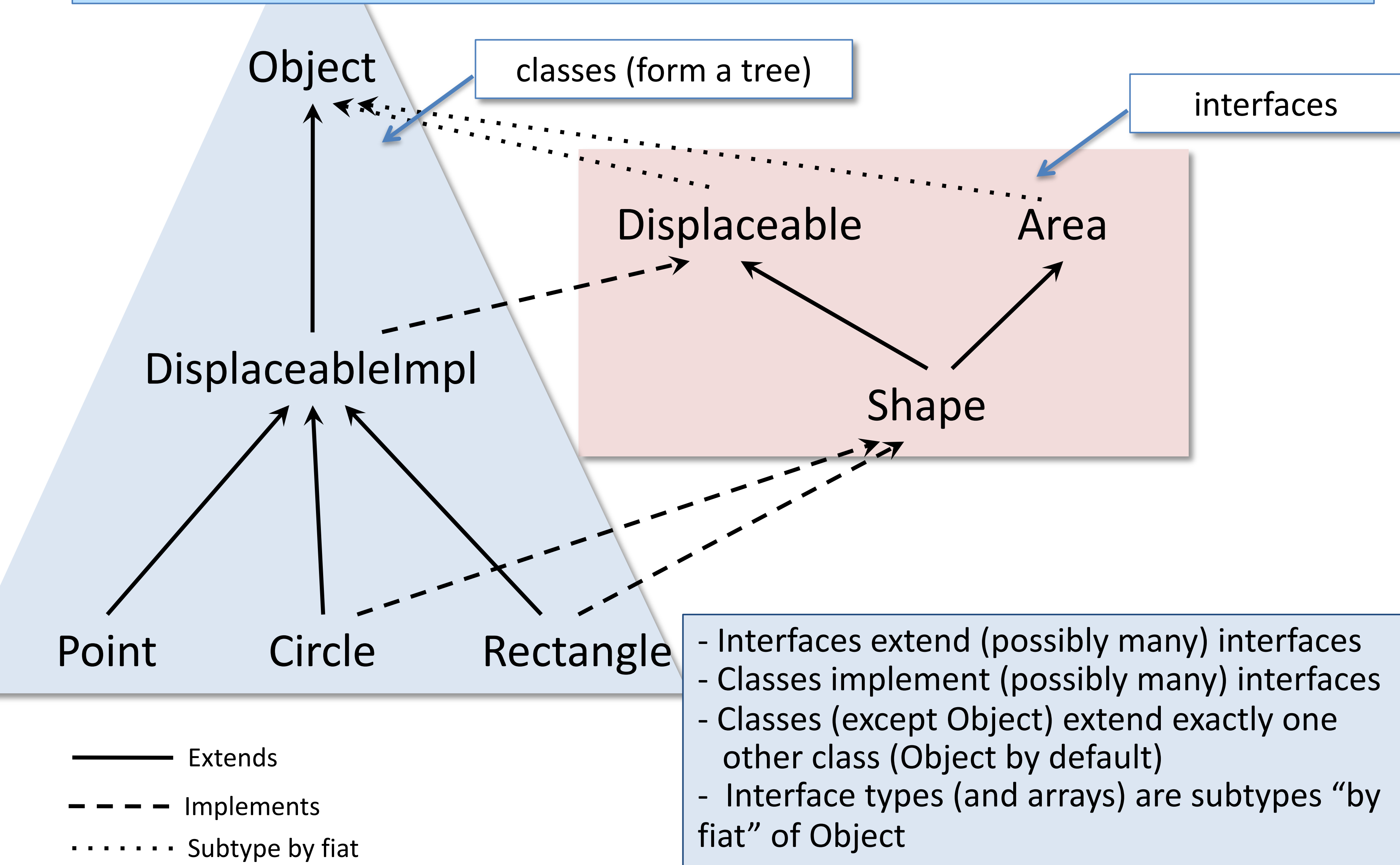
Note the “extends” keyword.

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 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).

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

Example of Simple Inheritance

See: Shapes.zip

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 (and a class)
  - 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%

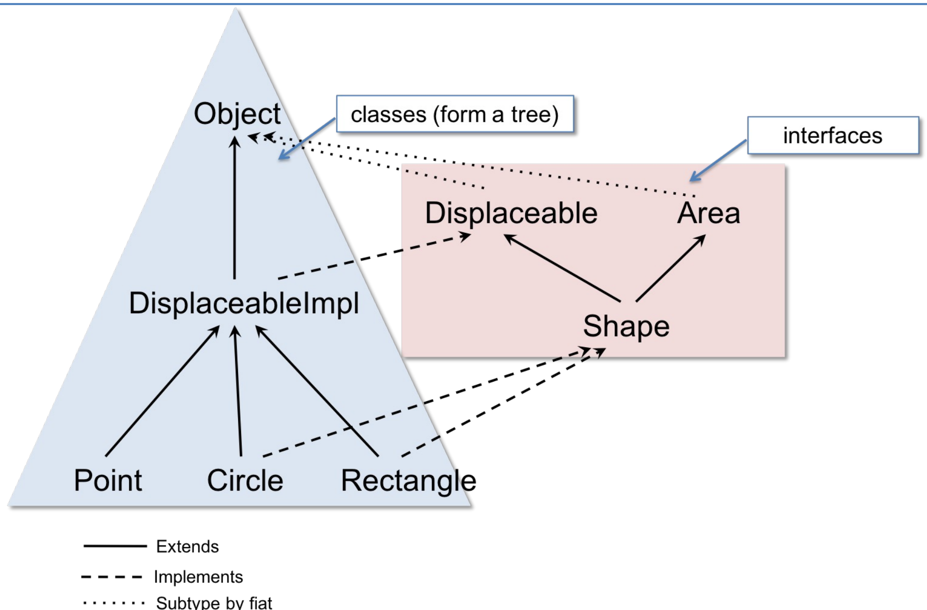
Not well typed — 0%

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

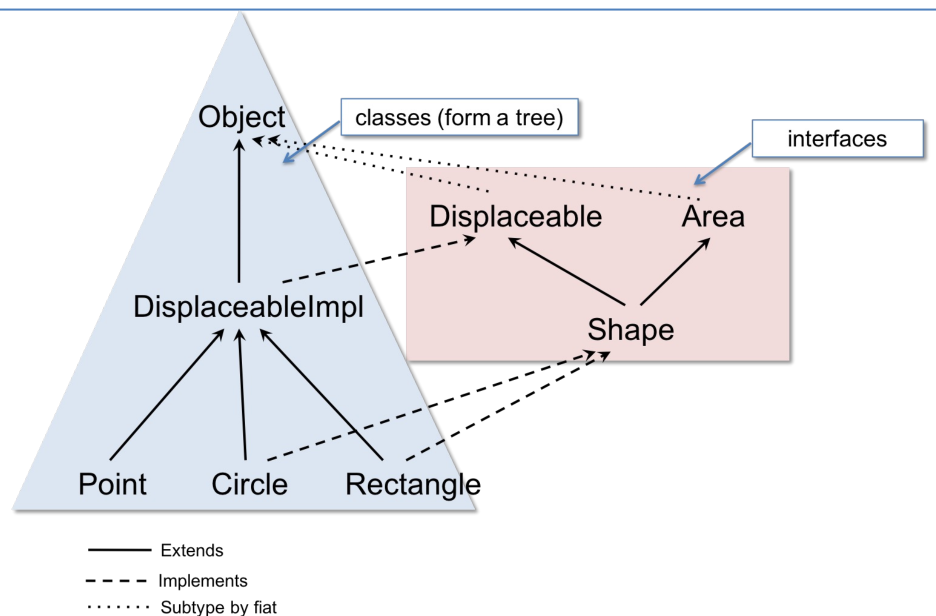
Area	0%
Circle	0%
None of the above	0%
Not well typed	0%

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

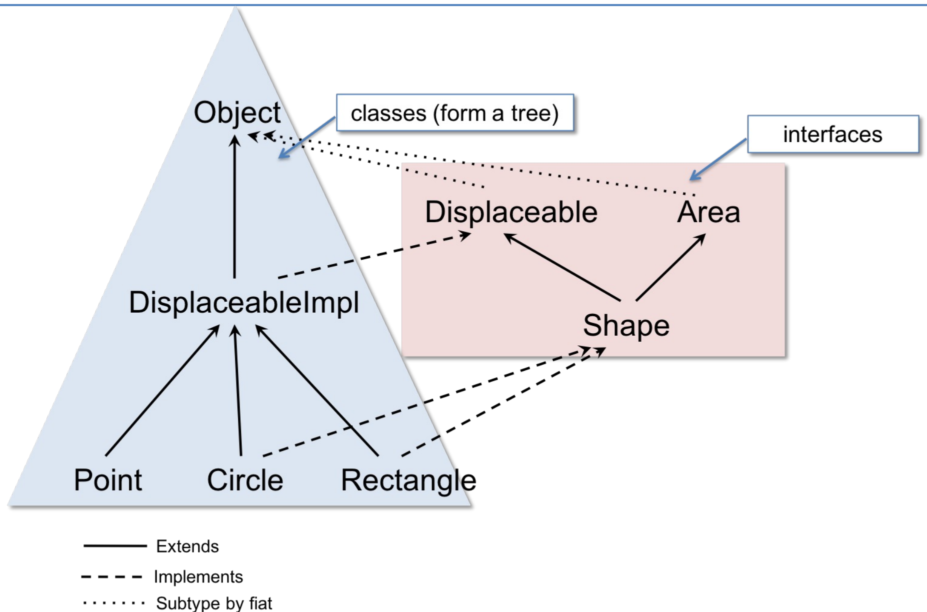
Area	0%
Circle	0%
None of the above	0%
Not well typed	0%

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 y (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’?

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;
   }
```

ASM refinement: The Class Table

Workspace

...

Stack

Heap

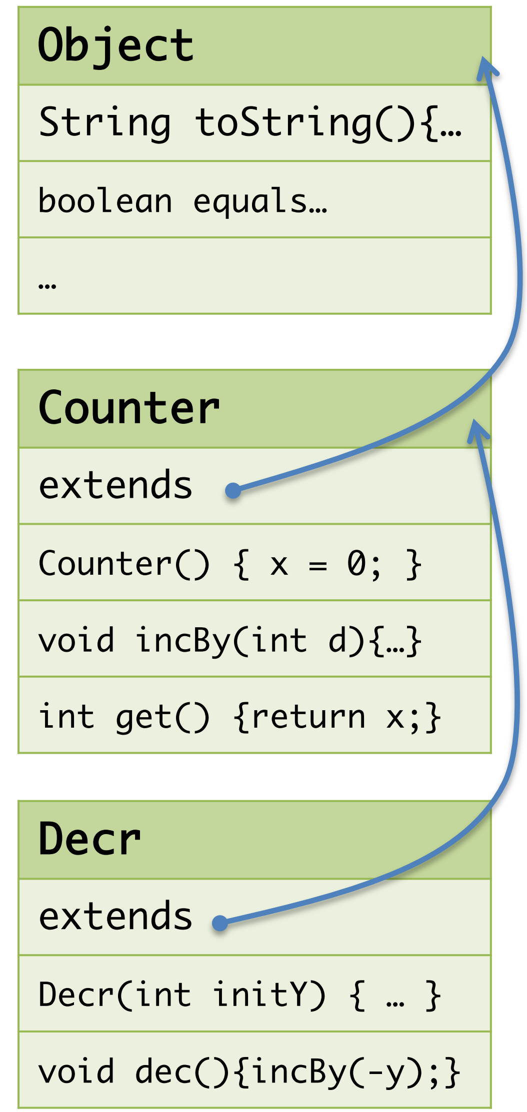
Class Table

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); }
}
```

Class Table



The class table contains:

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

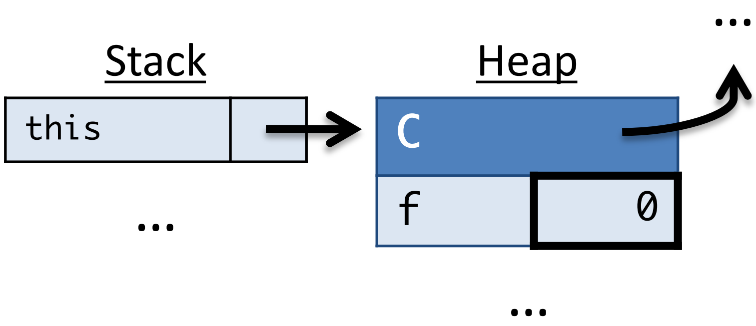
this

- 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 {
  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();
```

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

0

-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

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

Decr

- extends → Counter
- `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

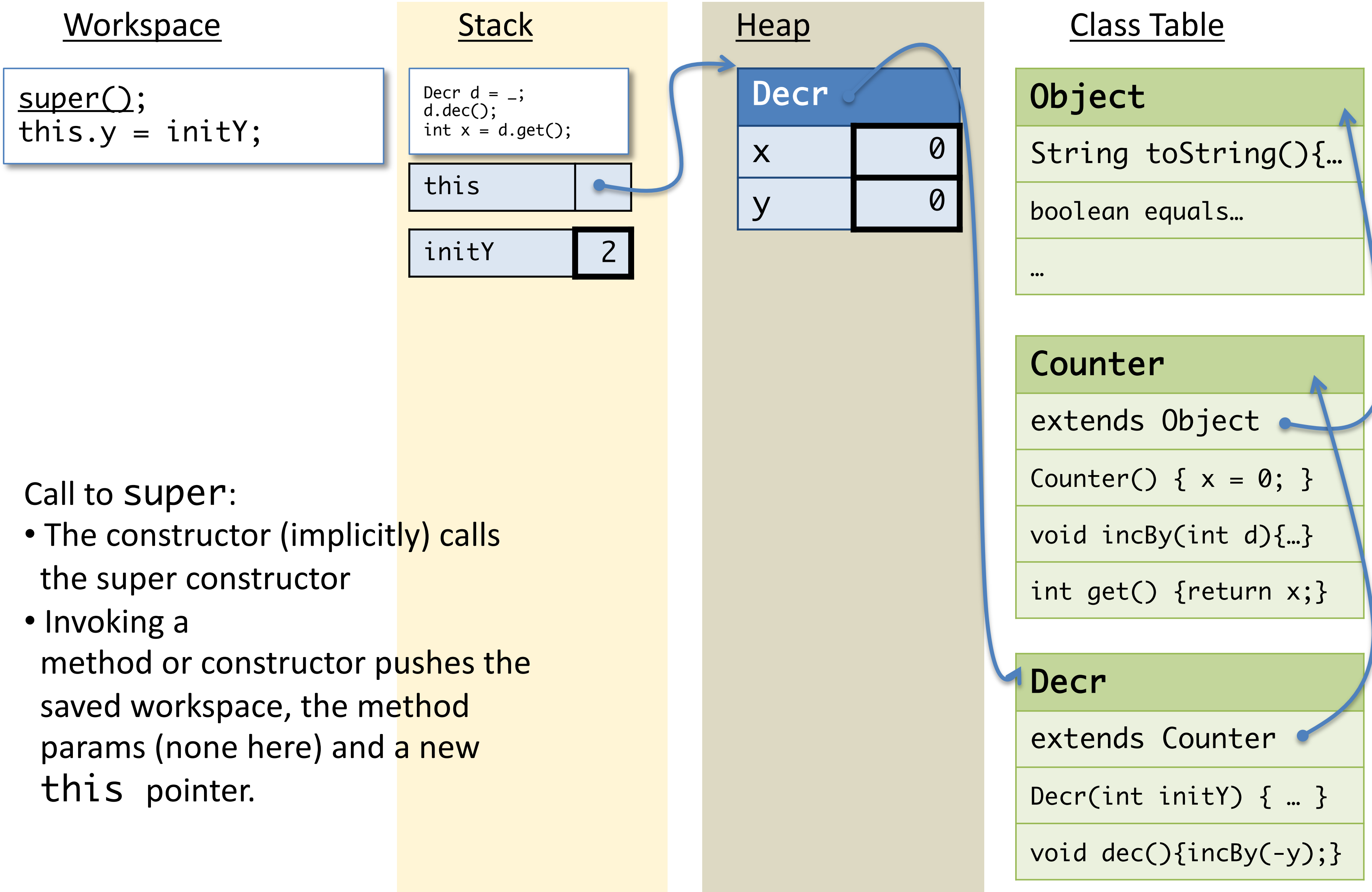
```
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
```

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

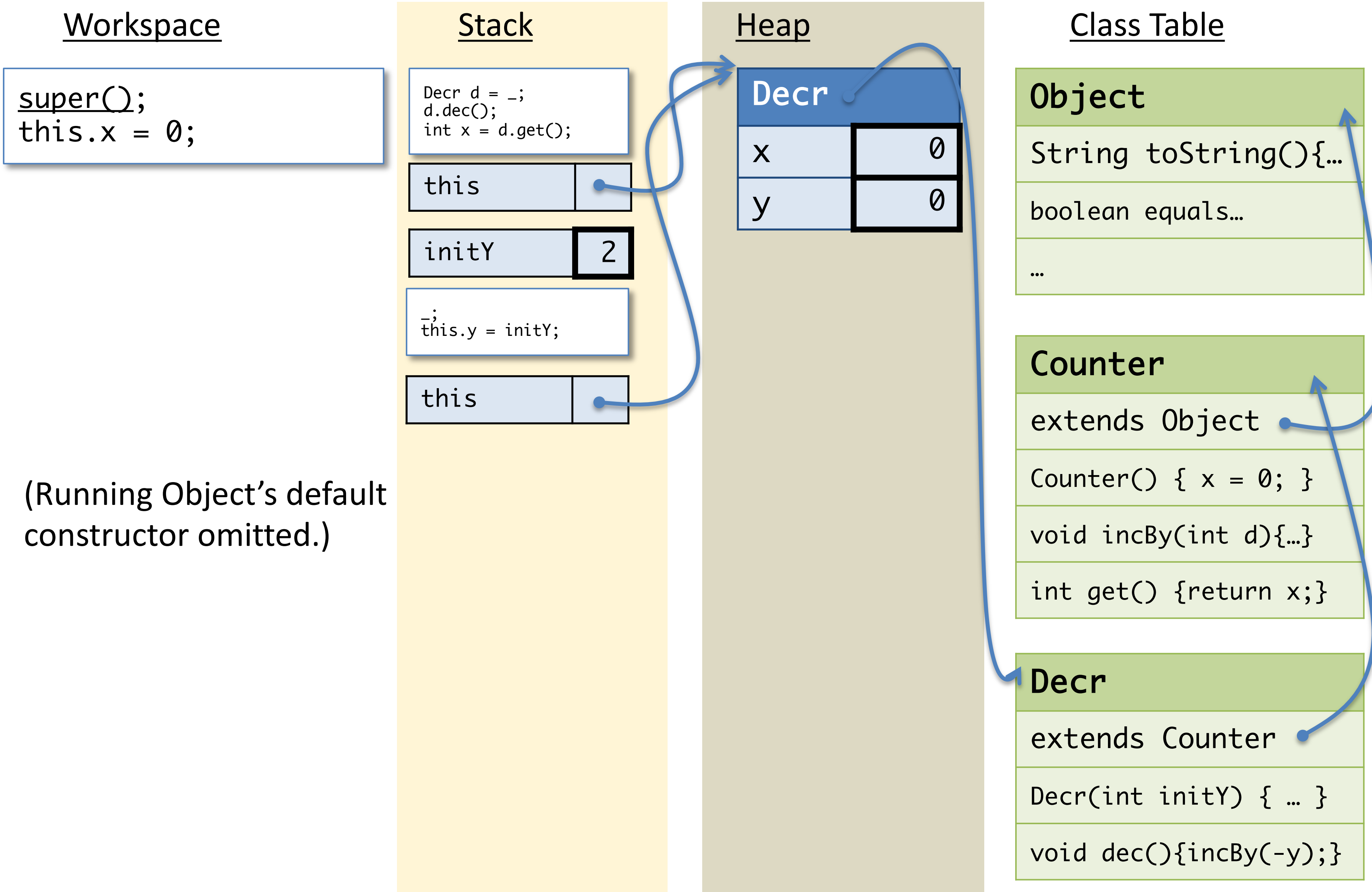
Calling super



Call to `super`:

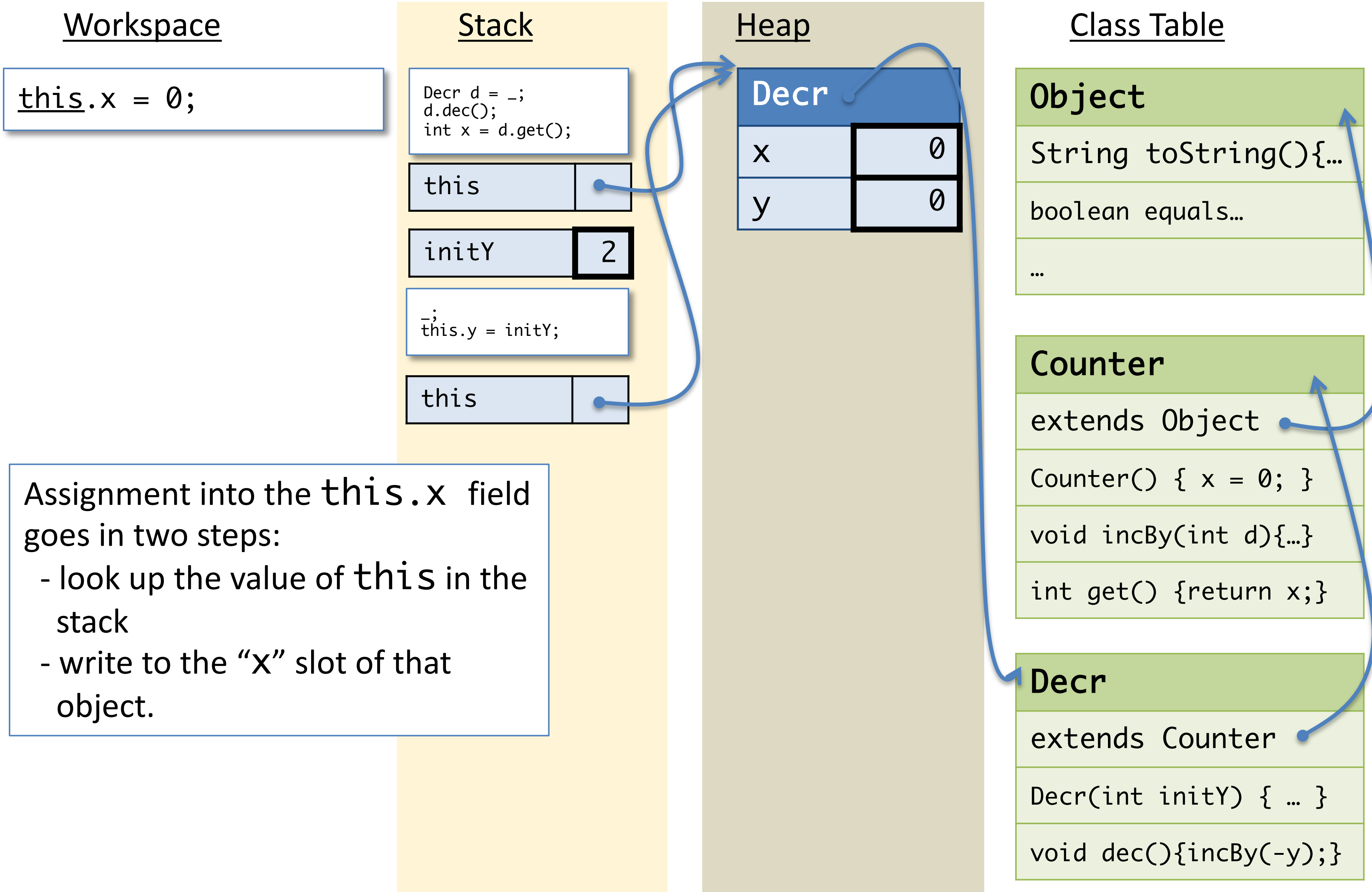
- The constructor (implicitly) calls the super constructor
- Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new `this` pointer.

Abstract Stack Machine



(Running Object's default constructor omitted.)

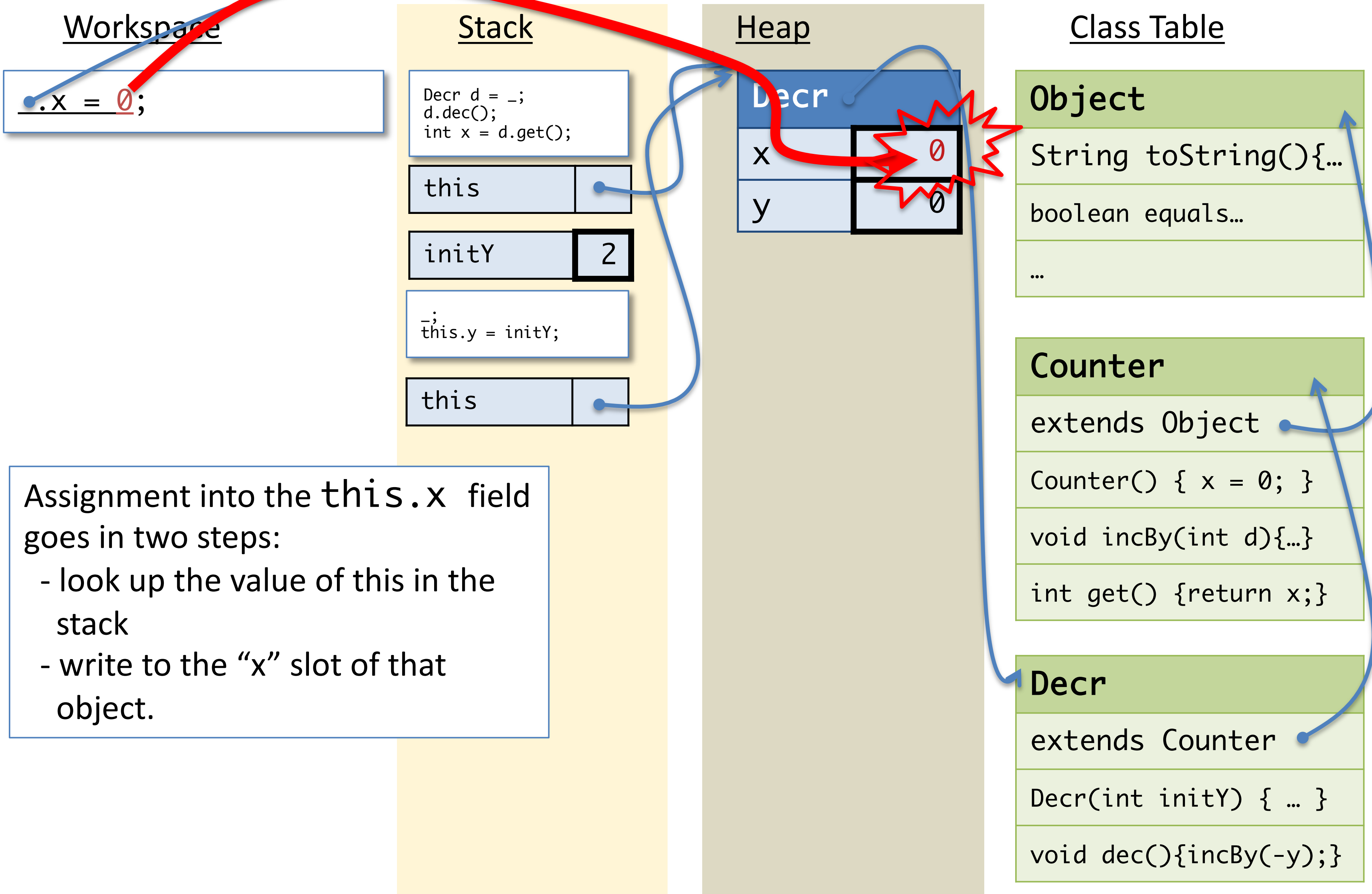
Assigning to a Field



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.

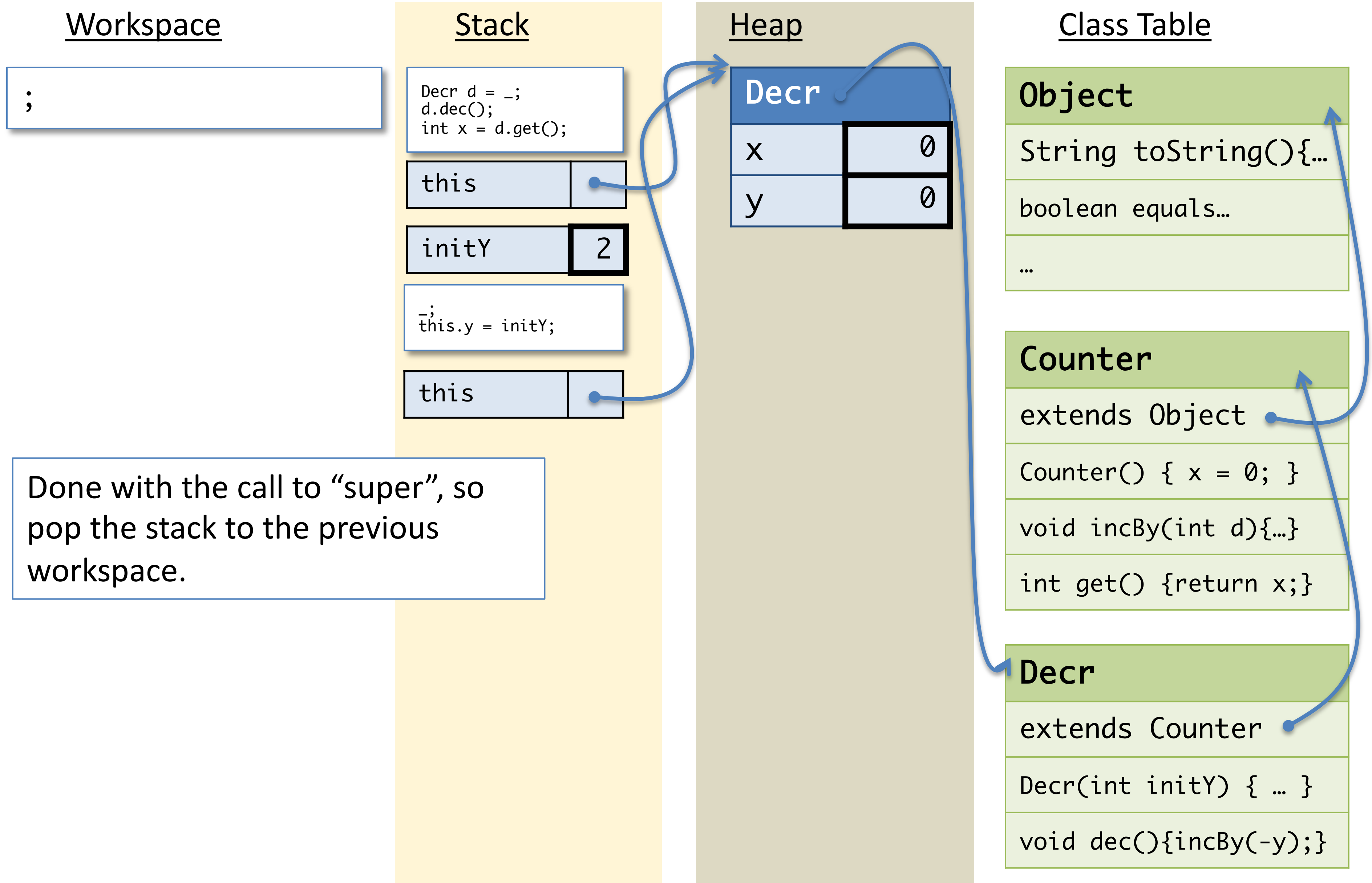
Assigning to a Field



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

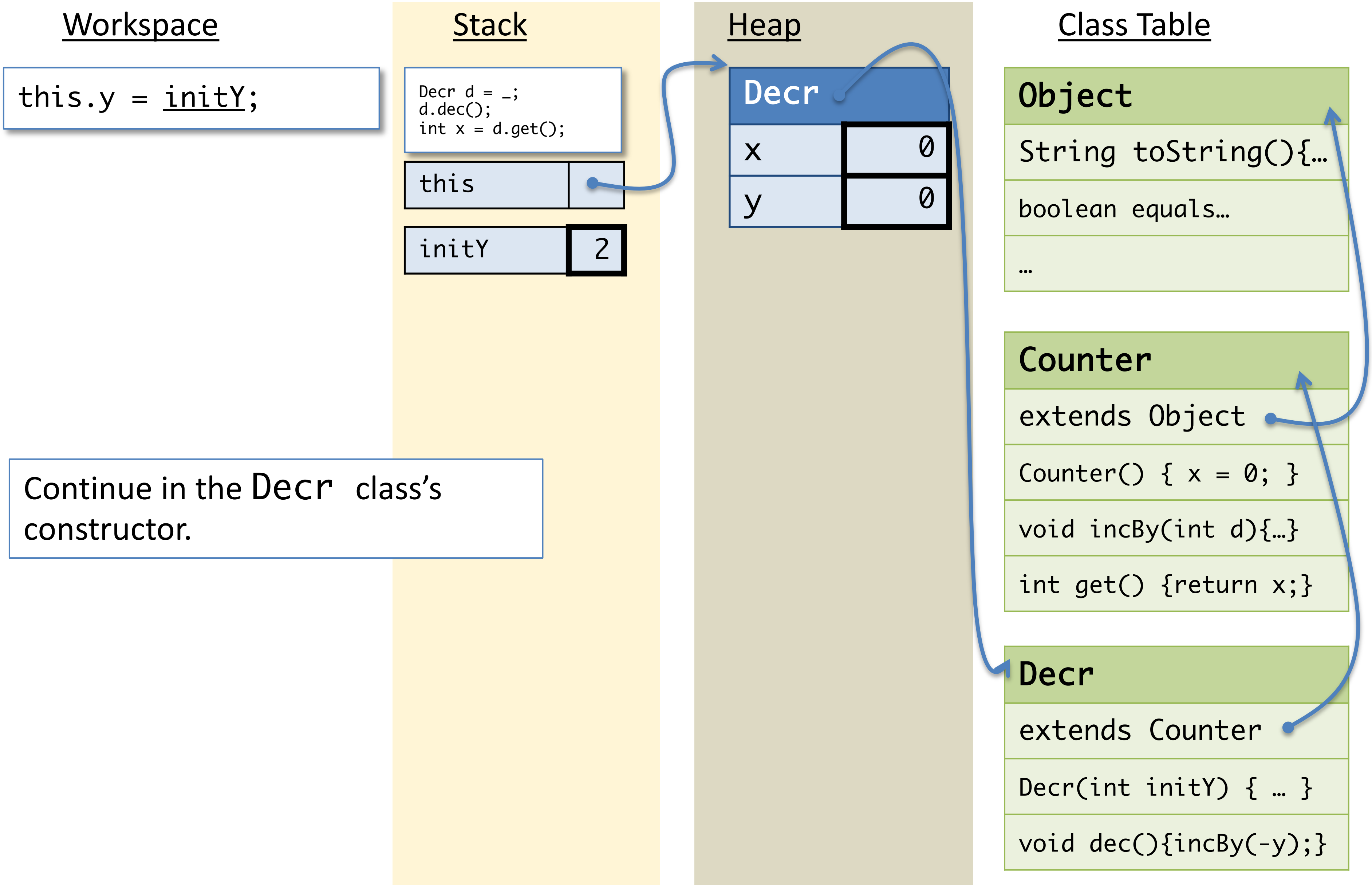
- look up the value of this in the stack
- write to the "x" slot of that object.

Done with the call



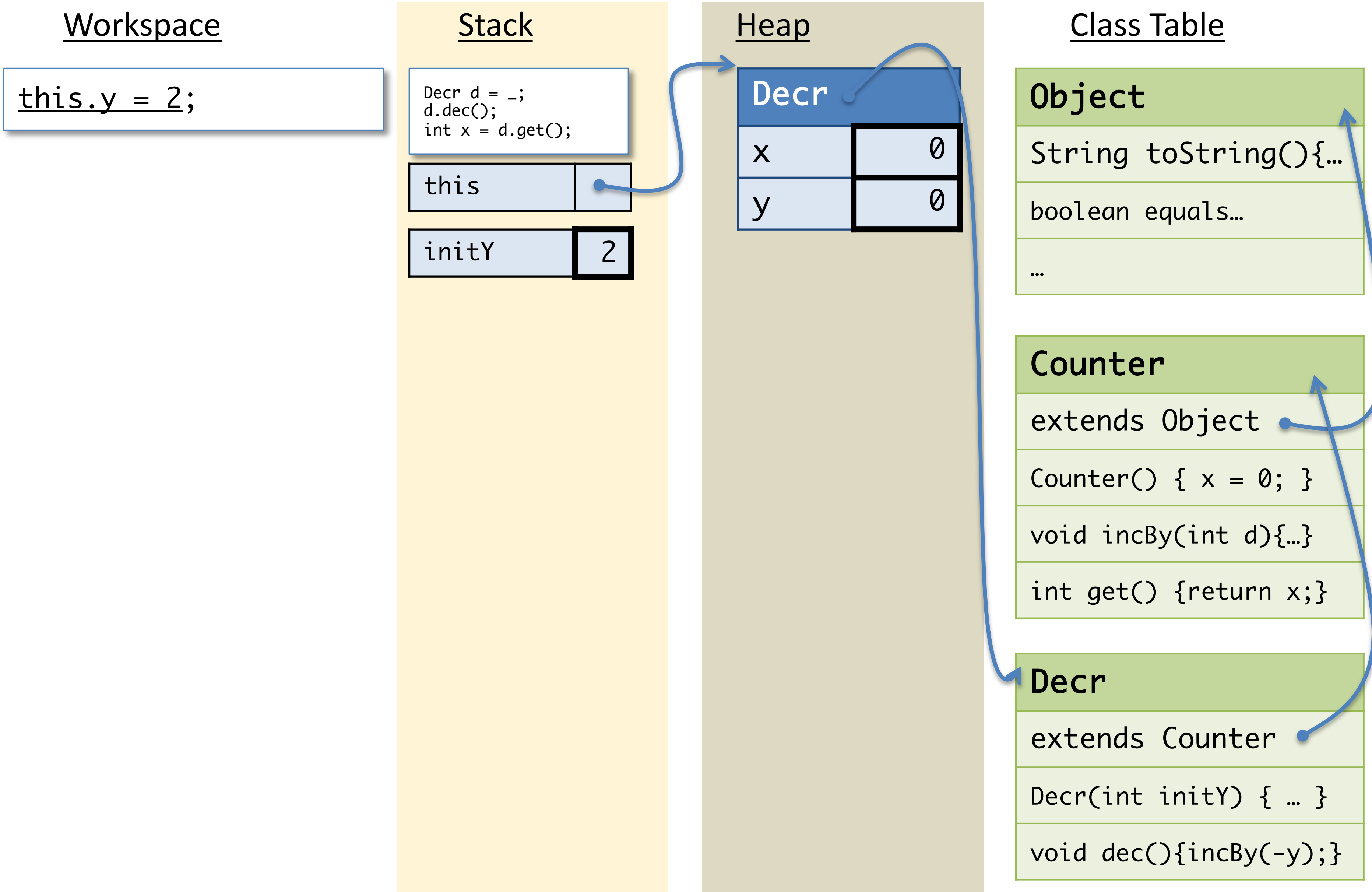
Done with the call to “super”, so pop the stack to the previous workspace.

Continuing

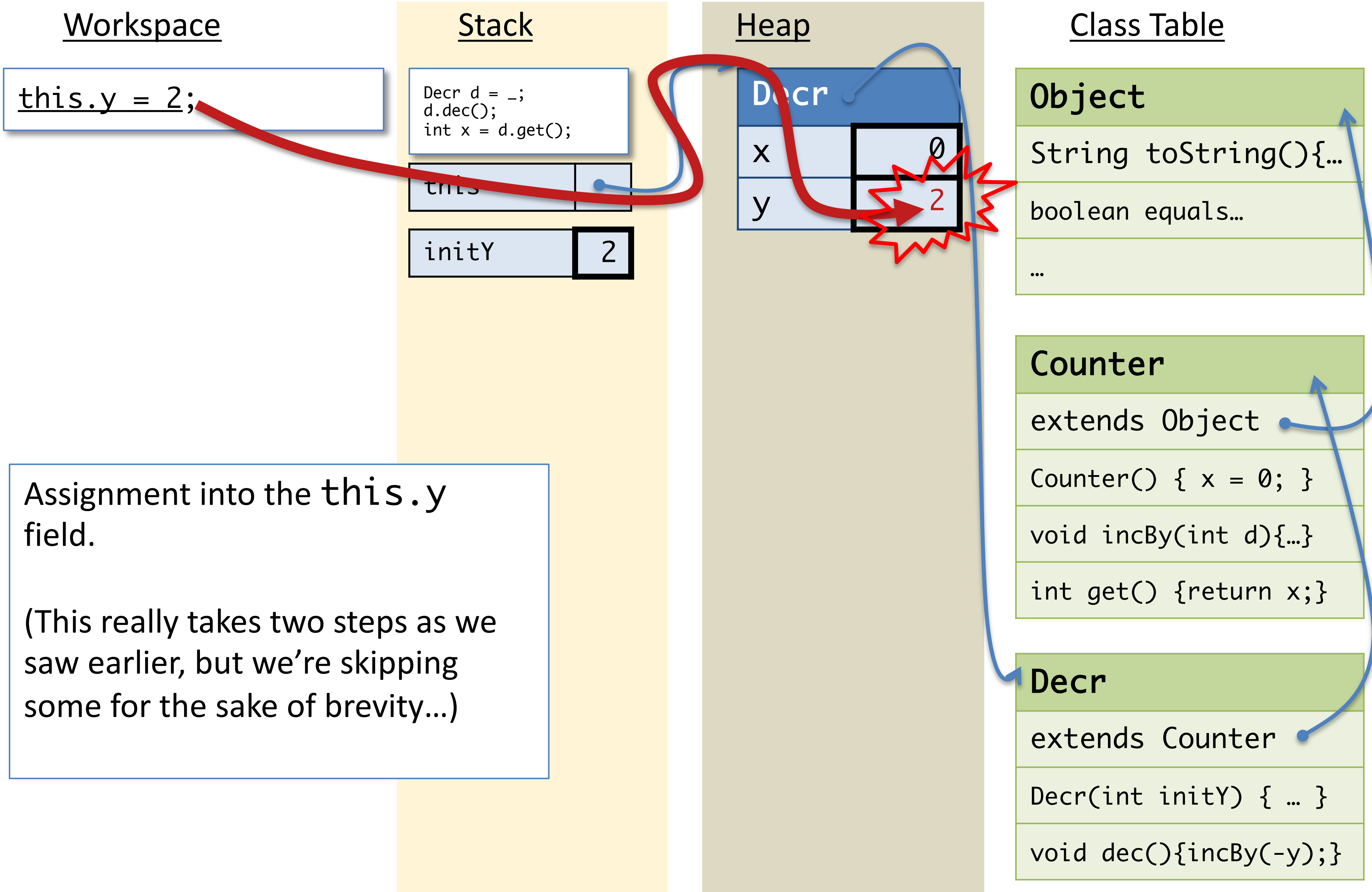


Continue in the Decr class's constructor.

Abstract Stack Machine



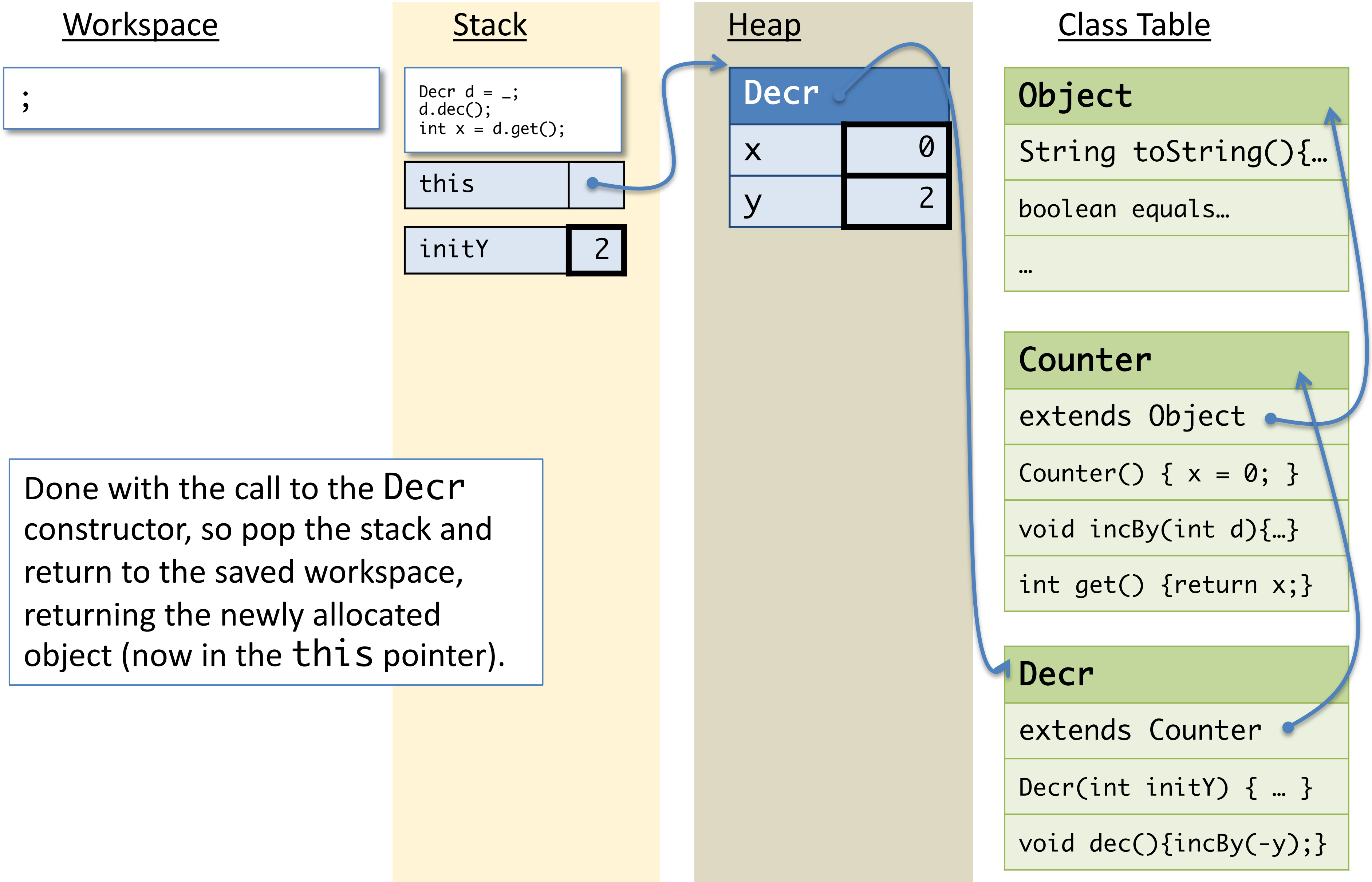
Assigning to a field



Assignment into the `this.y` field.

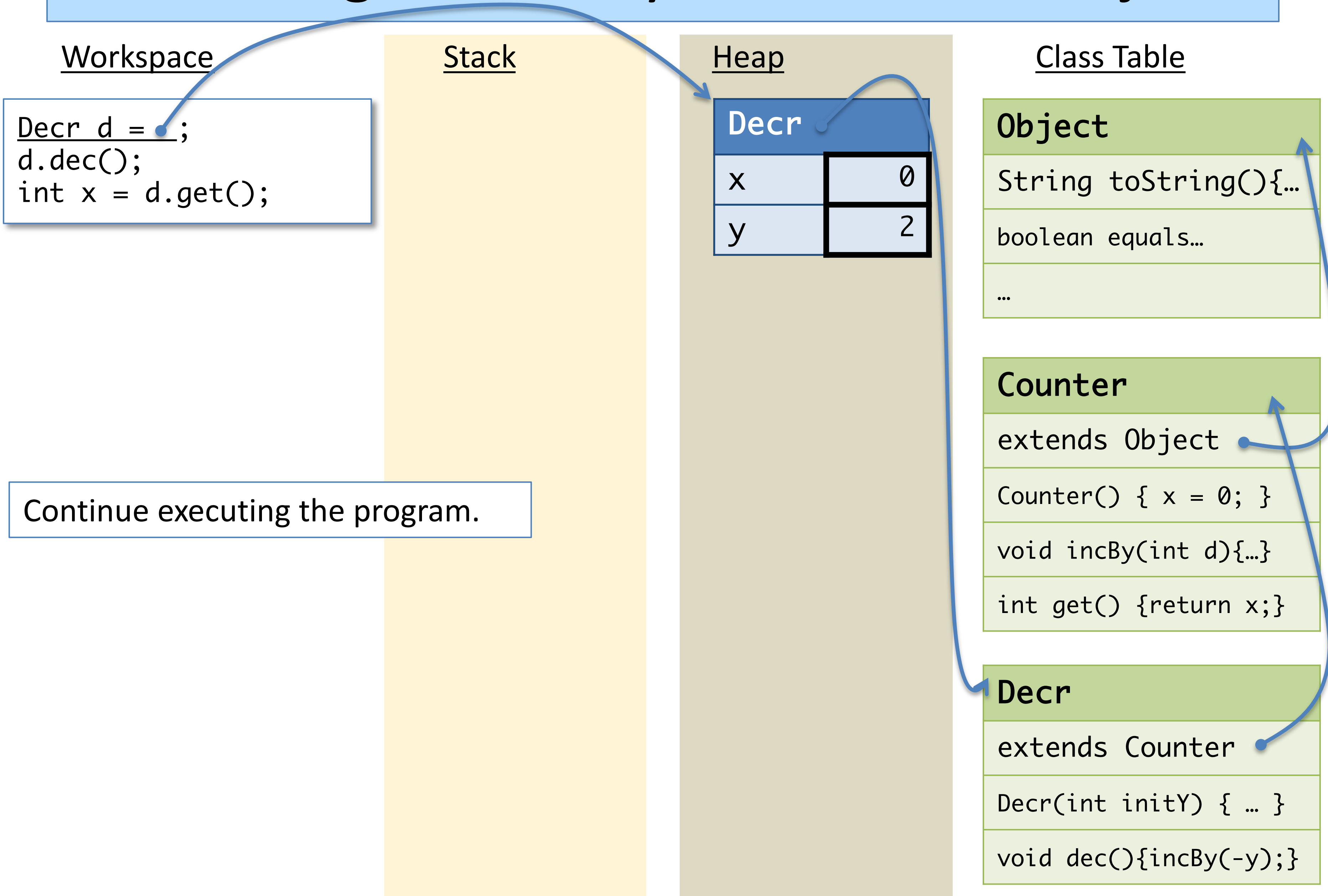
(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity…)

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).

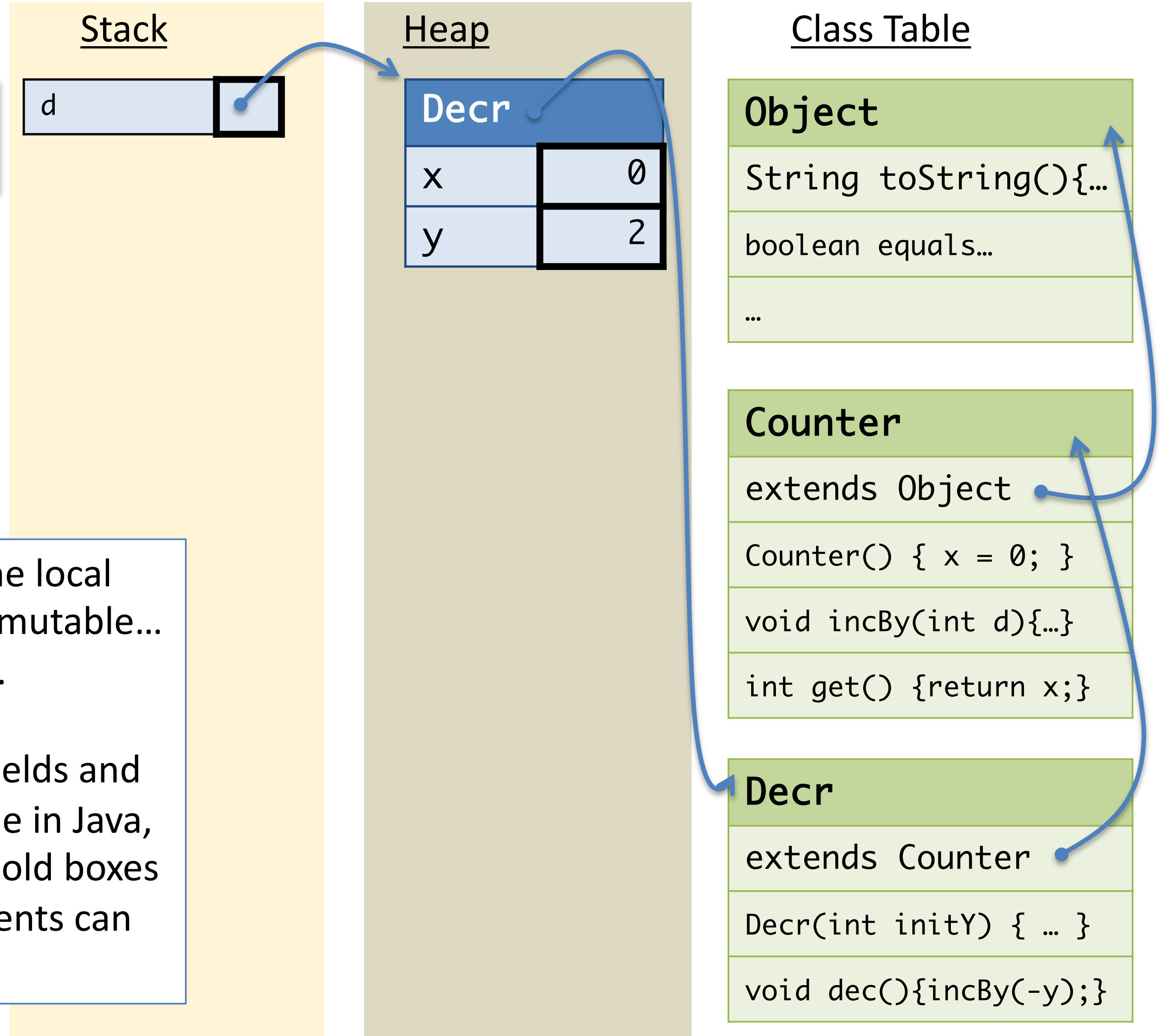
Returning the Newly Constructed Object



Allocating a local variable

Workspace

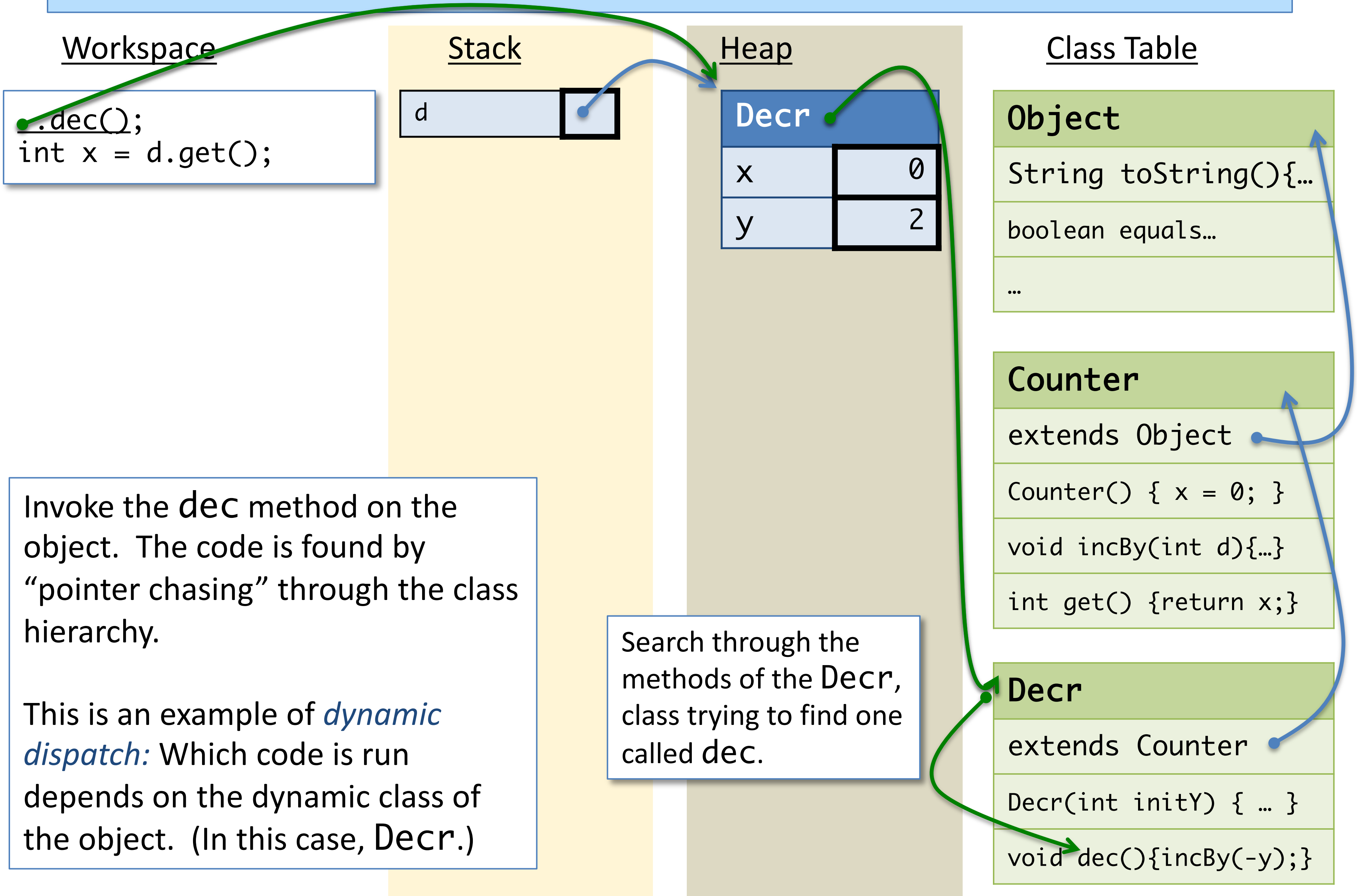
```
d.dec();
int x = d.get();
```



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

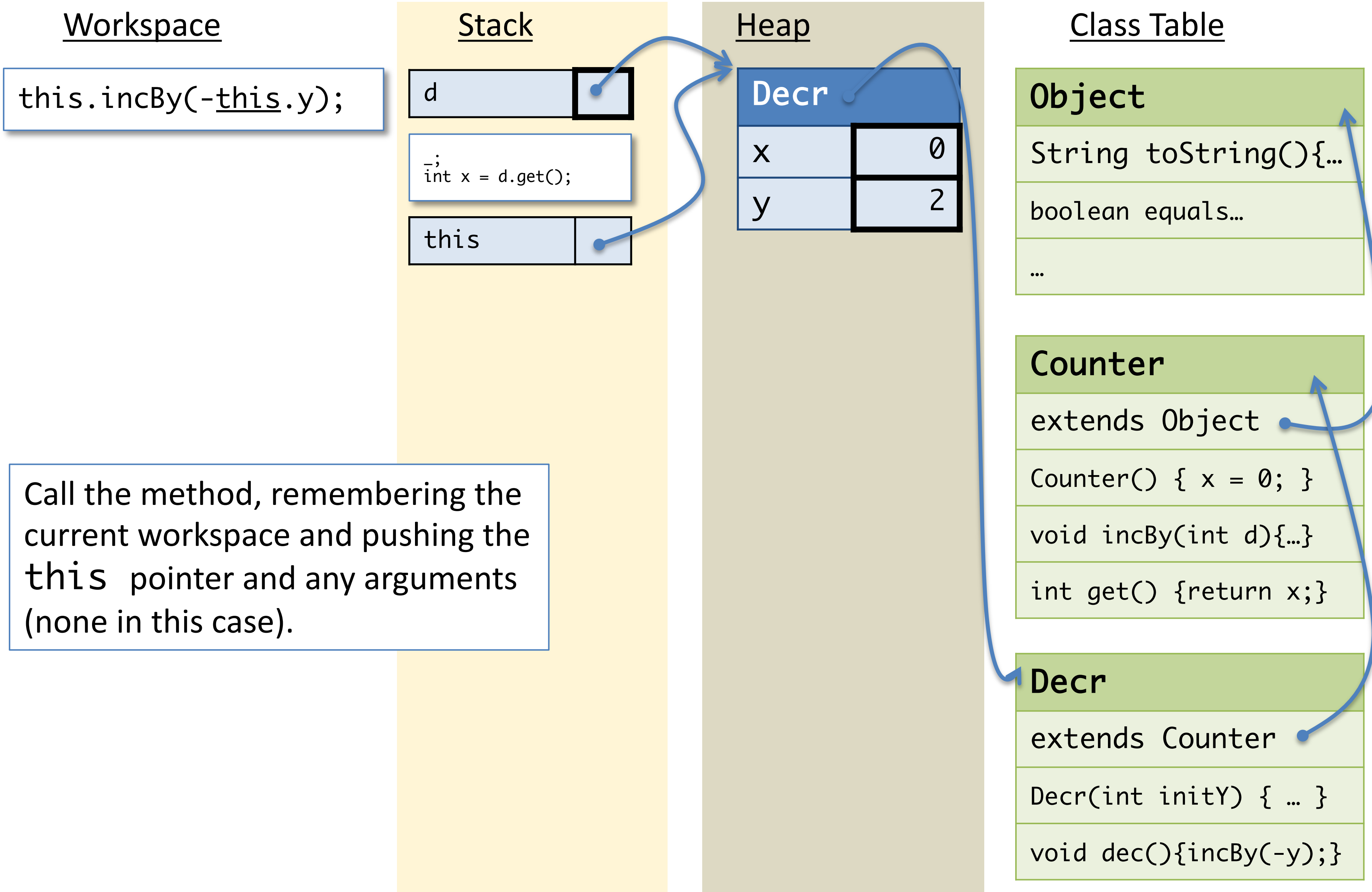


Invoke the dec method on the object. The code is found by "pointer chasing" through the class hierarchy.

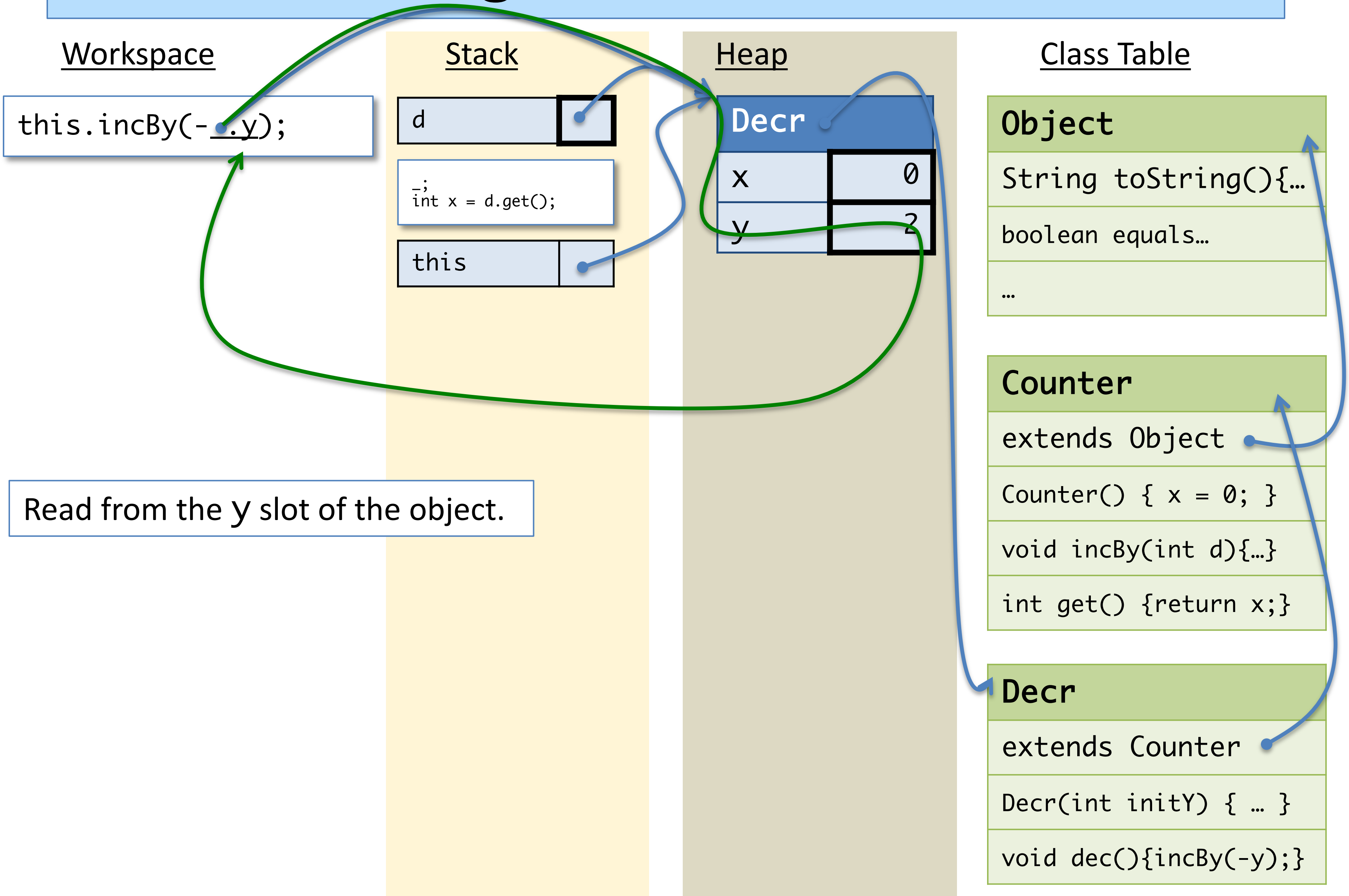
This is an example of *dynamic dispatch:* Which code is run depends on the dynamic class of the object. (In this case, Decr.)

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

Dynamic Dispatch: Finding the Code

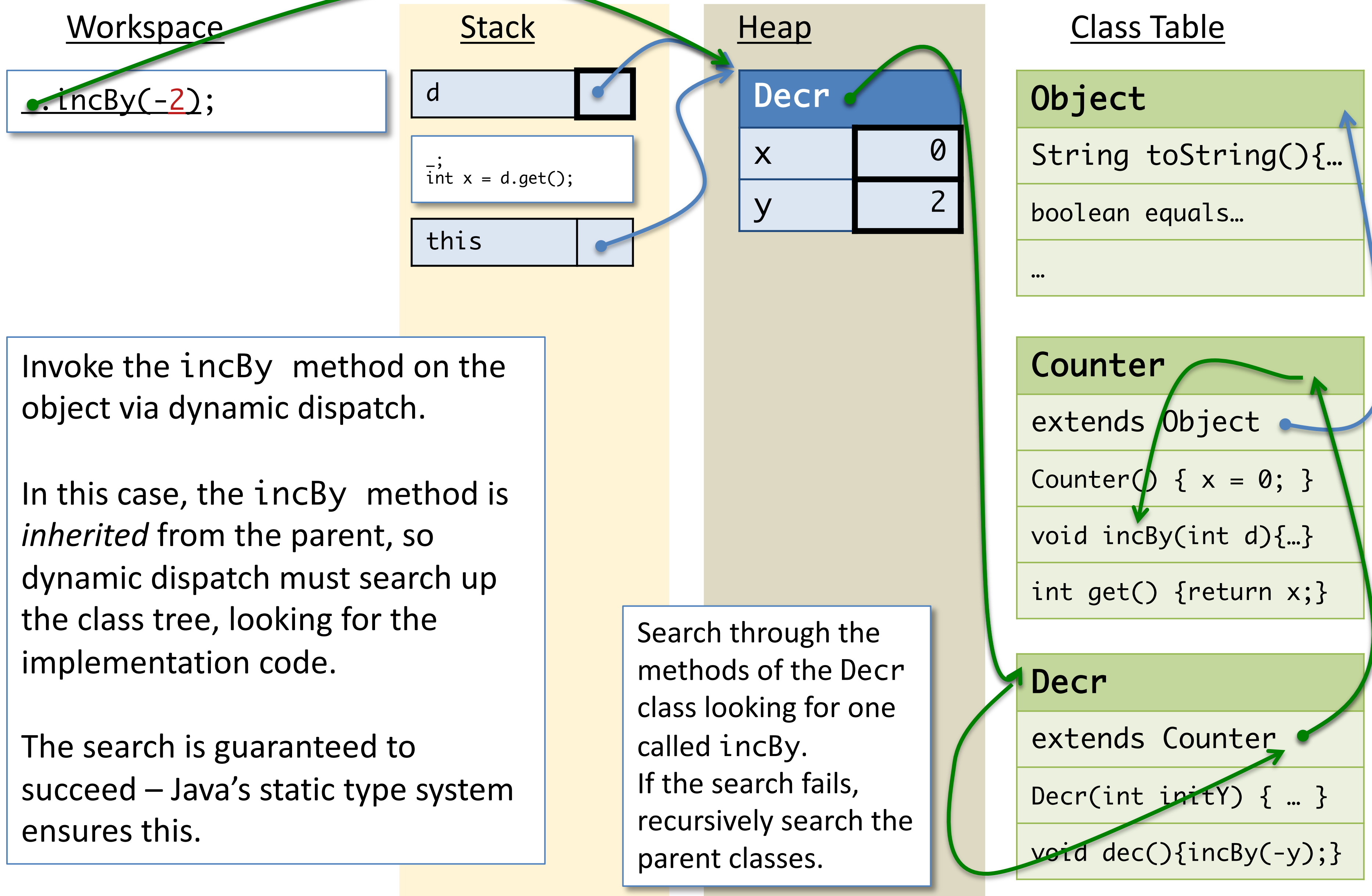


Reading a Field's Contents



Read from the y slot of the object.

Dynamic Dispatch, Again



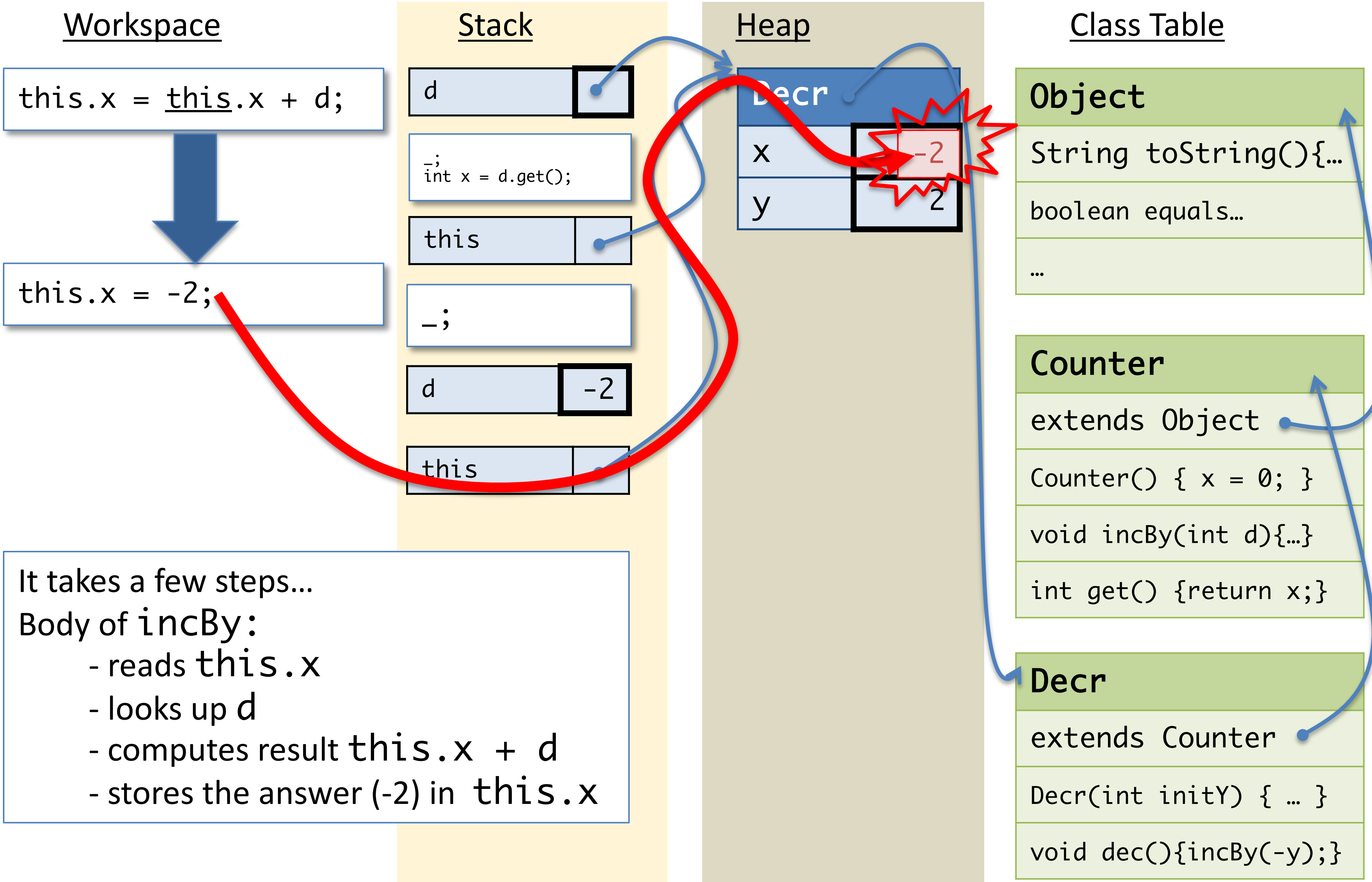
Invoke the `incBy` method on the object via dynamic dispatch.

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

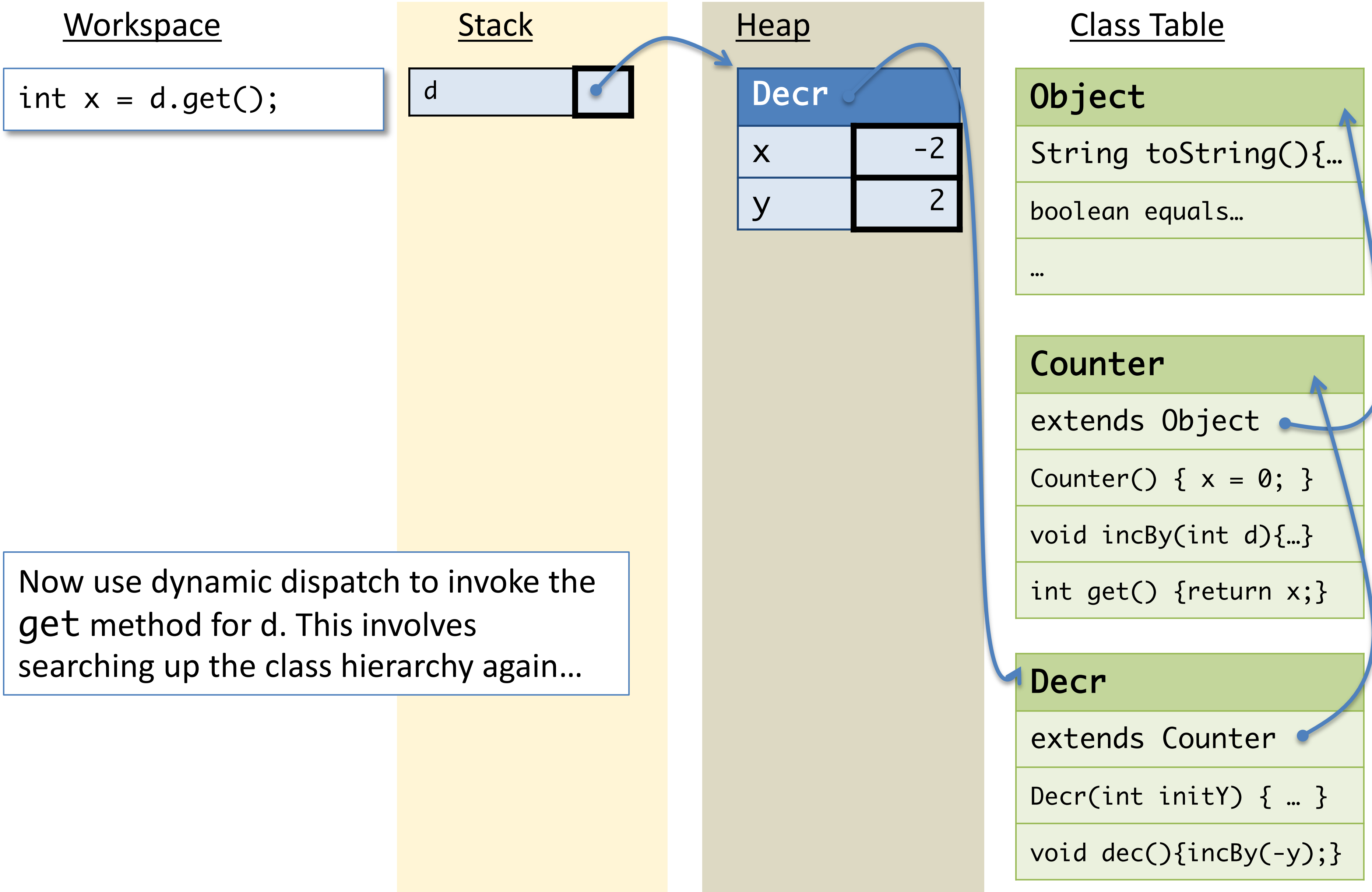


It takes a few steps…

Body of `incBy`:

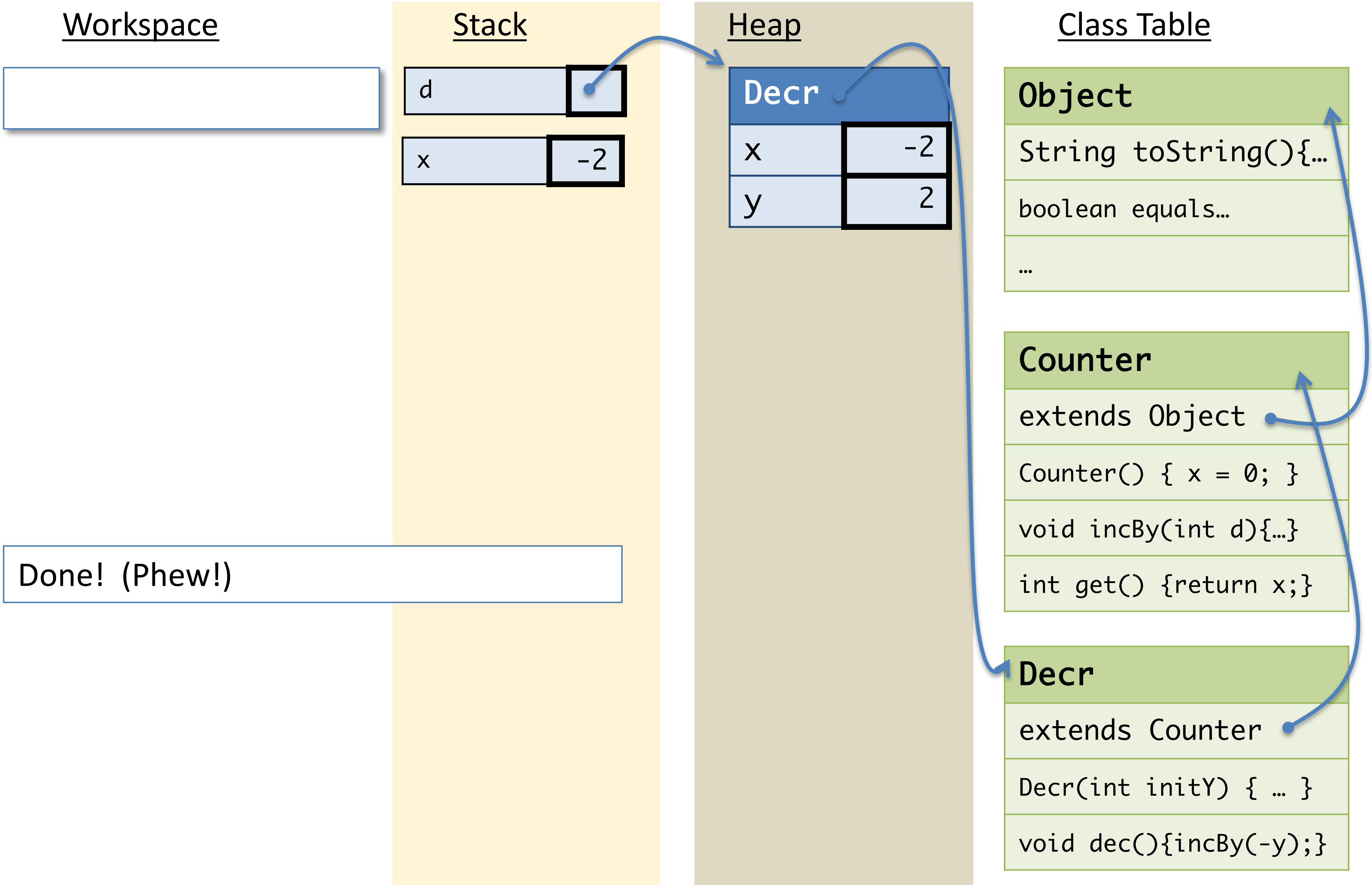
- reads `this.x`
- looks up `d`
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

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 reference 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 references 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` reference 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()`

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

0

No

0%

Yes

0%

I'm not sure

0%

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

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` reference
  - 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

Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism

Review: Subtype Polymorphism*

- Main idea:

> Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

- If B is a subtype of A, it provides all of A's (public) methods.

*polymorphism = many shapes

Is subtype polymorphism enough?

Mutable Queue Interface in OCaml

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the front value and return it (if any) *)
  val deq : 'a queue -> ‘a

  (* Determine if the queue is empty *)
  val is_empty : ‘a queue -> bool
end
```

How can we translate this interface to Java?

Java Interface using Subtyping

```ocaml
module type QUEUE =
sig

  type 'a queue

  val create : unit -> 'a queue

  val enq : 'a -> 'a queue -> unit

  val deq : 'a queue -> 'a

  val is_empty : 'a queue -> bool

end
```

OCaml

```java
interface ObjQueue {



   // no constructors
   // in an interface

    public void enq(Object elt);

    public Object deq();

    public boolean isEmpty();

}
```

Java

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
__A__  x = q.deq();
```

What type should we write for A?

1. String
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Subtype Polymorphism

```
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
System.out.println(x.trim());
```

trim is a method of the String class (removes extra spaces)

← Does this line type check

1. Yes
2. No
3. It depends

ANSWER: No

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Parametric Polymorphism (a.k.a. Generics)

- Main idea:

Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {
  void enq(E o);
  E deq();
  boolean isEmpty();
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  - the implementation of enq cannot invoke any methods on ‘o’ (except those inherited from Object)
  - i.e., the only thing we know about E is that it is a subtype of Object

Generics (Parametric Polymorphism)

```
public interface Queue<E> {
  void enq(E o);
  E deq();
  boolean isEmpty();
  …
}
```

```
Queue<String> q = …;

q.enq(" CIS 120 ");
String x = q.deq();              // What type of x?    String
System.out.println(x.trim());    // Is this valid?     Yes!
q.enq(new Point(0.0,0.0));       // Is this valid?     No!
```

Subtyping and Generics

Subtyping and Generics*

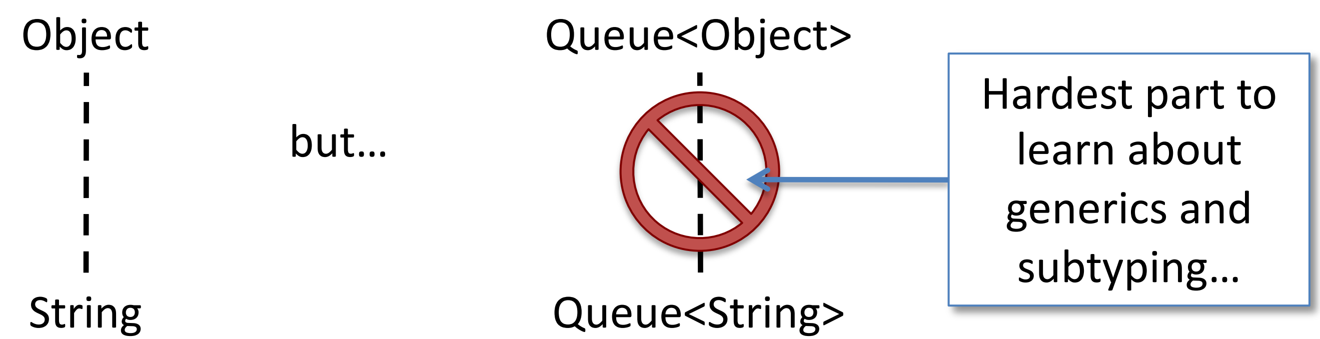
```
Queue<String> qs = new QueueImpl<>();
Queue<Object> qo = qs;

qo.enq(new Object());
String s = qs.deq();
```

Ok? Sure!
Ok? Let's see…

Ok? I guess
Ok? Noooo!

- Java generics are *invariant*:
  - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:

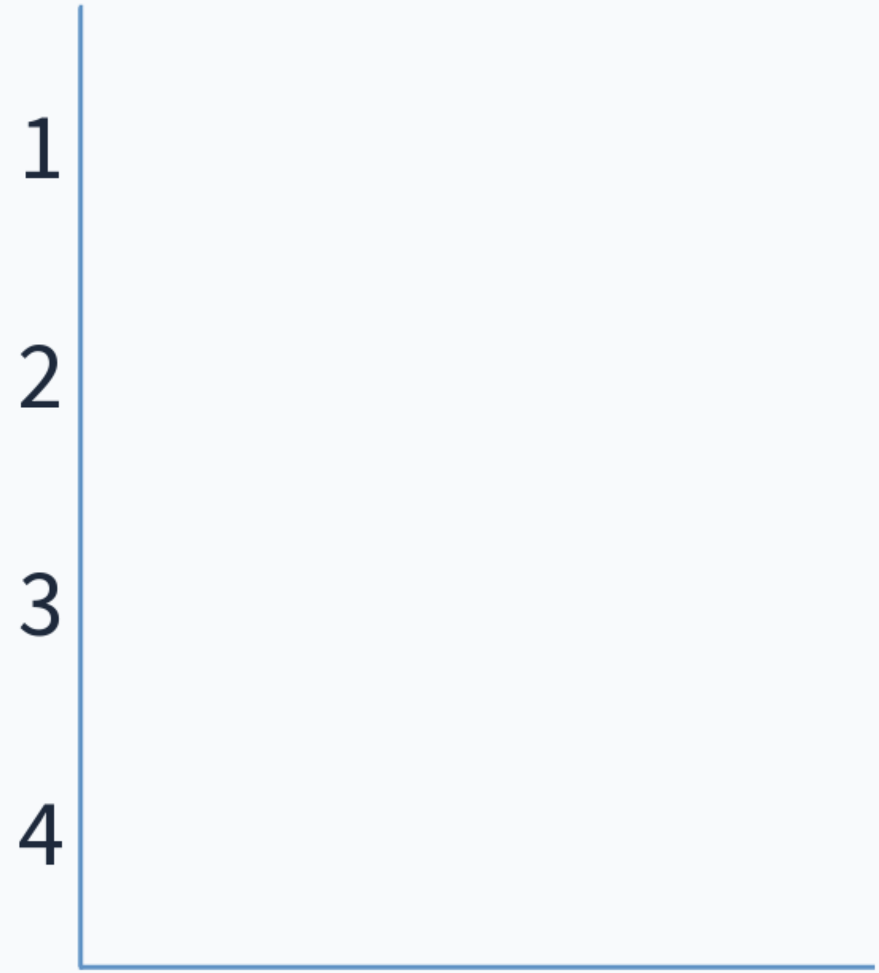


* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.

26: Subtyping with Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither



Subtyping and Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

Answer: 1