Programming Languages and Techniques (CIS120)

Lecture 25

Mar 19, 2012

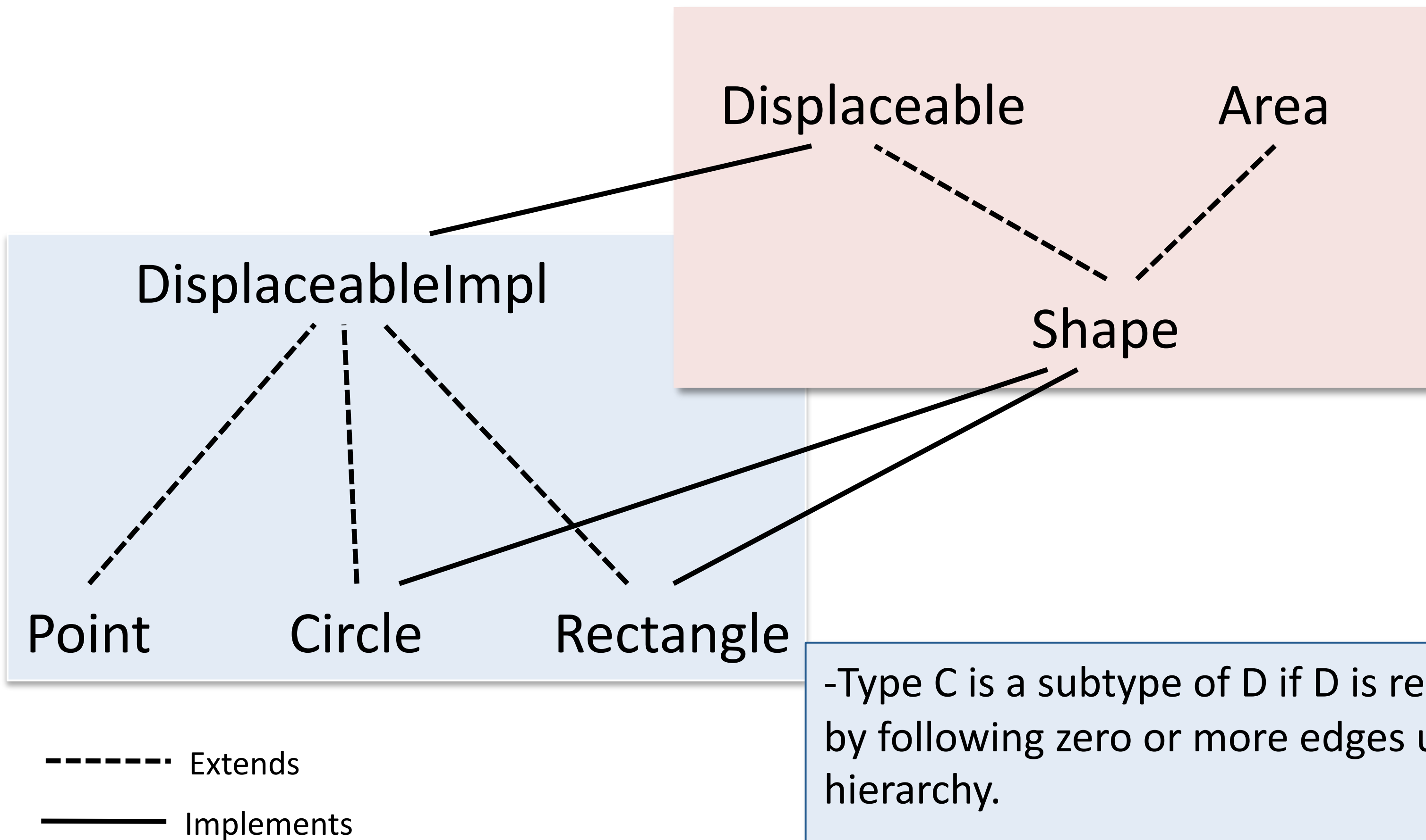
Subtyping and Dynamic Dispatch

Announcements

- HW08 due next Monday
  - Come by office hours today if you don’t have Java 6 installed
  - Remember to use .equals for string comparison

Interface Extension
& Class Inheritance

Subtyping with Inheritance



-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

Inheritance: Constructors

- Contructors *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).

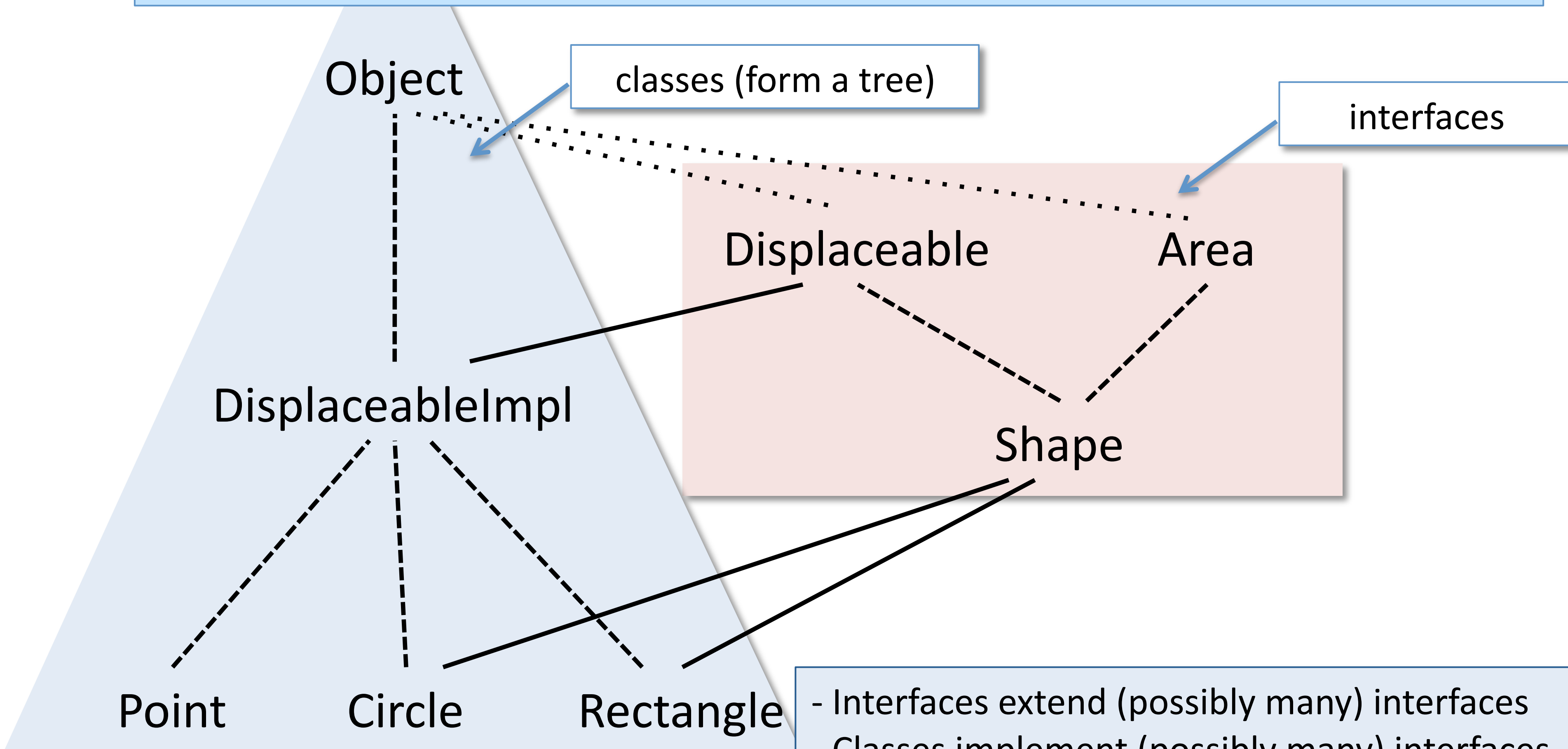
```
class D {
  private int x;
  private int y;
  public D (int initX, int initY) { x = initX; y = initY; }
  public int addBoth() { return x + y; }
}

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); }
}
```

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 that leave off the “extends” clause *implicitly* extend Object
  - Arrays also implement the methods of Object
  - This class provides methods useful for *all* objects to support (override these!)
- Object is also the top type of the subtyping hierarchy.

Subtyping



- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object if implicit)
- Interface types 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 hard to use properly and the need for them arises in 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 (see Main3.java).

  Especially avoid overriding.

The Java Abstract Stack Machine

1. Class tables
2. Constructors and "this"
3. Dynamic dispatch
4. Static class members

What about nonstatic methods?

- 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 about inheritance?

Refinements to the Stack Machine

- Code is stored in a *class table*, which is a special part of the heap:
  - When a program starts, the JVM initializes the class table
  - Each class has a pointer to its (unique) parent in the class tree
  - A class stores the constructor and method code for its instances
  - The class also stores *static* members

- Constructors:
  - Allocate space in the heap
  - (Implicitly) invoke the super class constructor, then run the constructor body

- Objects and their methods:
  - Each object in the heap has a pointer to the class table of its dynamic (the one it was created with via `new`).
  - A method invocation "`o.m(…)`" uses `o`'s class table to "dispatch" to the appropriate method code (might involve searching up the class hierarchy).
  - Methods and constructors take an implicit "`this`" parameter, which is a pointer to the object whose method was invoked. Fields& methods are accessed with `this.`

The ‘`this`’ Reference

- Inside a non-static method, the variable `this` is a reference to the object itself.
- 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();
```

Example 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

<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);}

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…
…

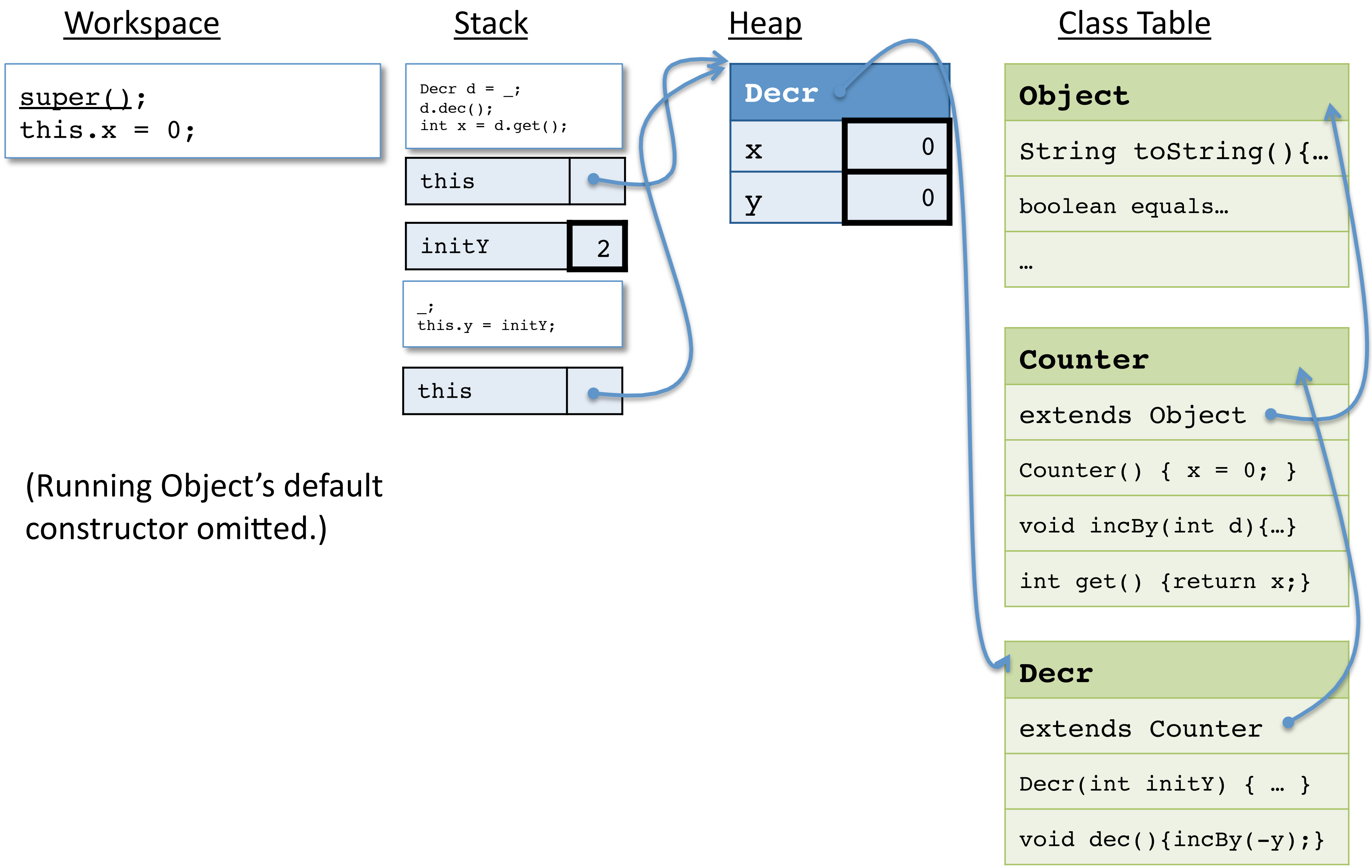
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}

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

Call to super:

- The constructor (implicitly) calls the super constructor
- Remember that invoking a method/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

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

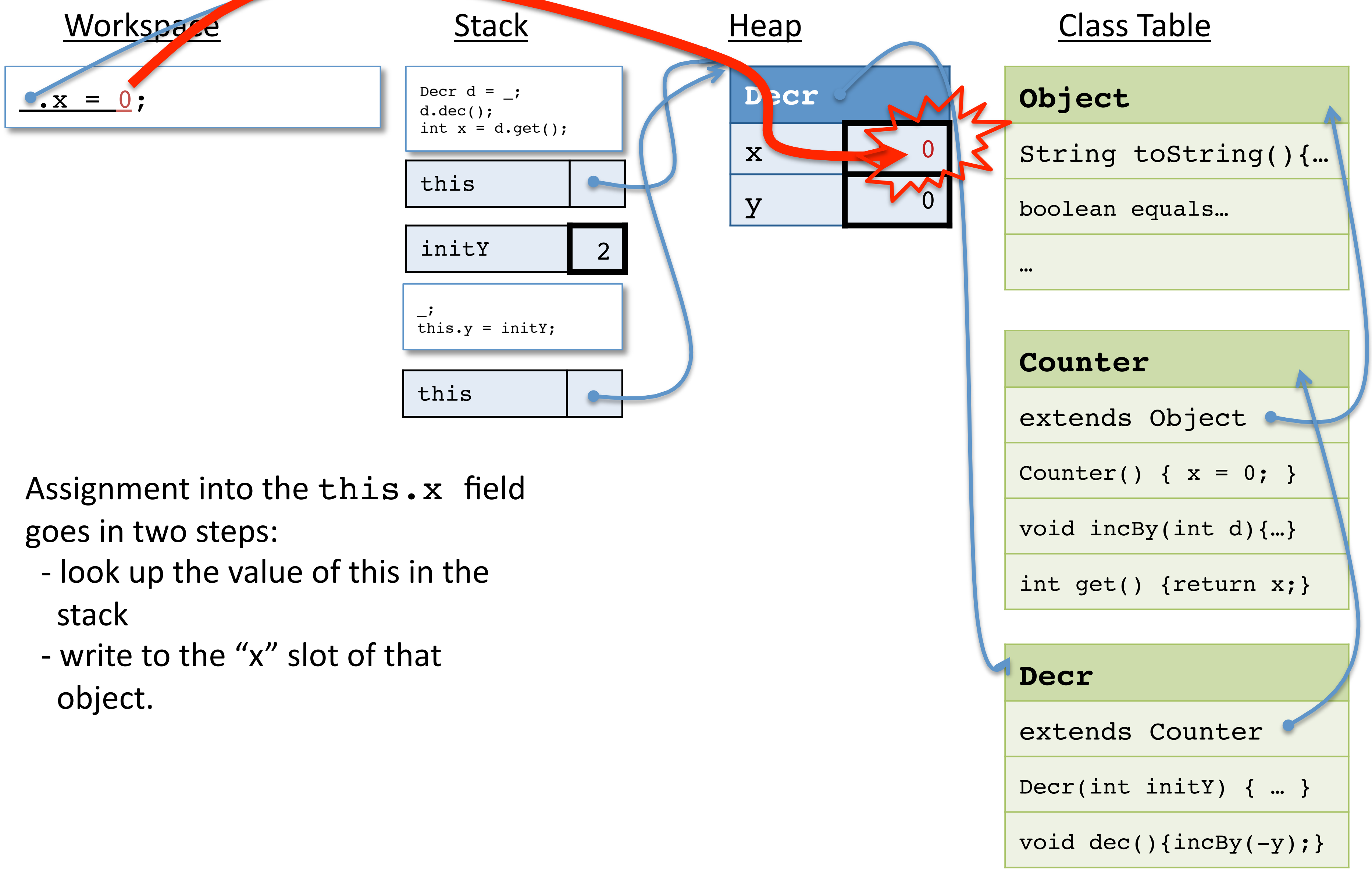
```
Object
String toString(){…
boolean equals…
…
```

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

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

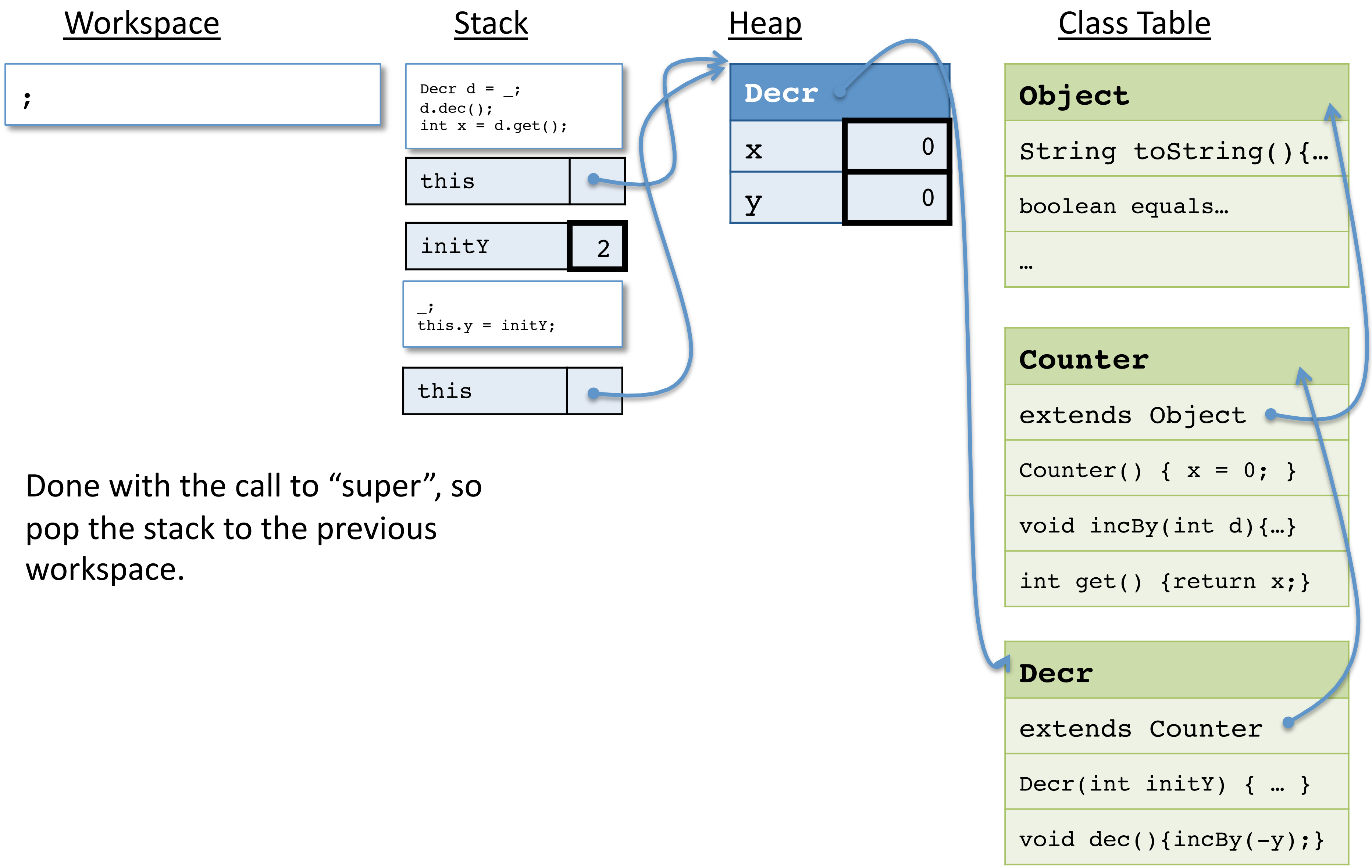
Assignment into the `this.x` field goes in two steps:
- look up the value of this in the stack
- write to the “x” slot of that object.

Assigning to a Field



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

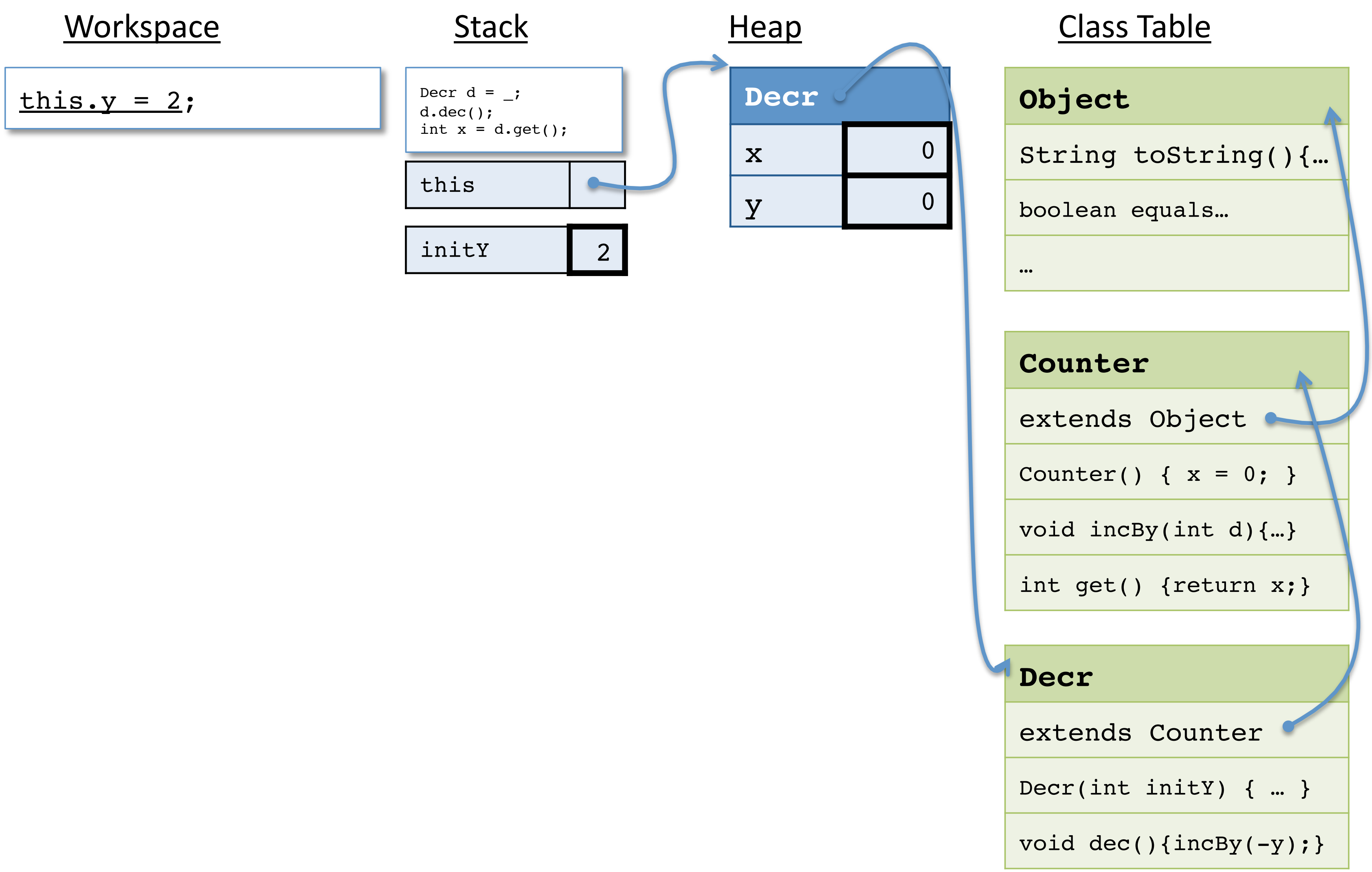
Object
String toString(){…
boolean equals…
…

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

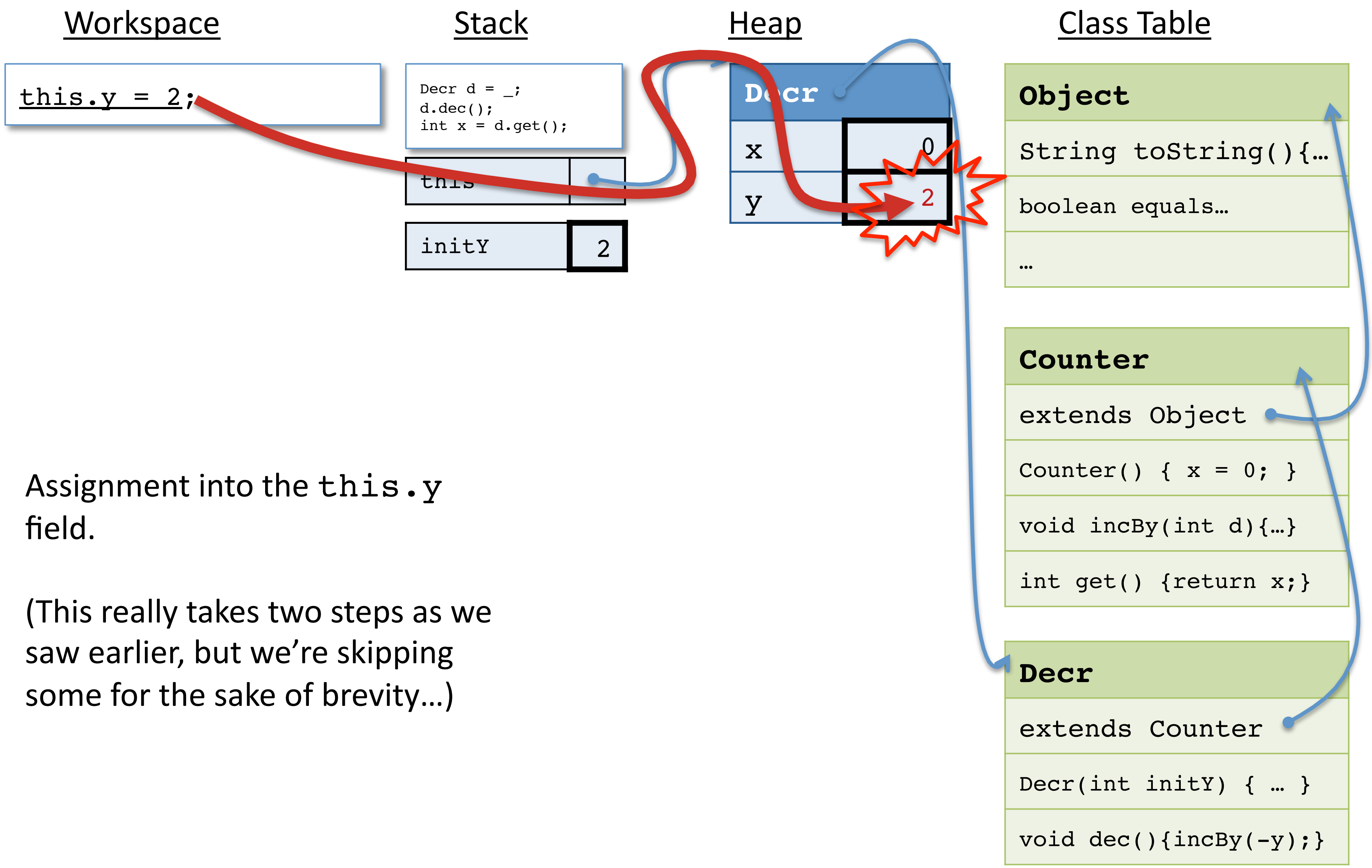
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}

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

Workspace

```
;
```

Stack

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

this	
initY	2

Heap

Decr	
x	0
y	2

Class Table

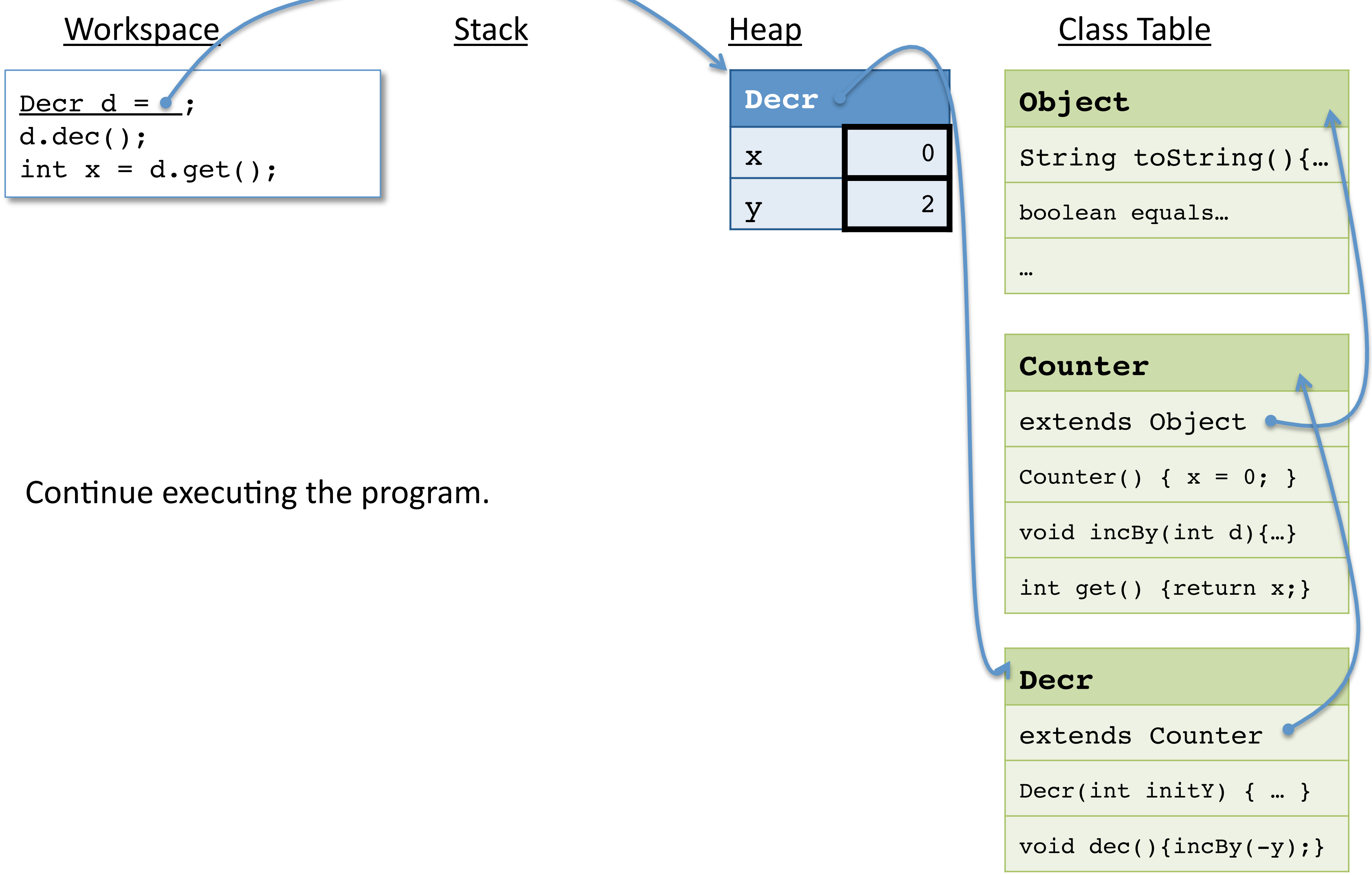
Object
String toString(){…
boolean equals…
…

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

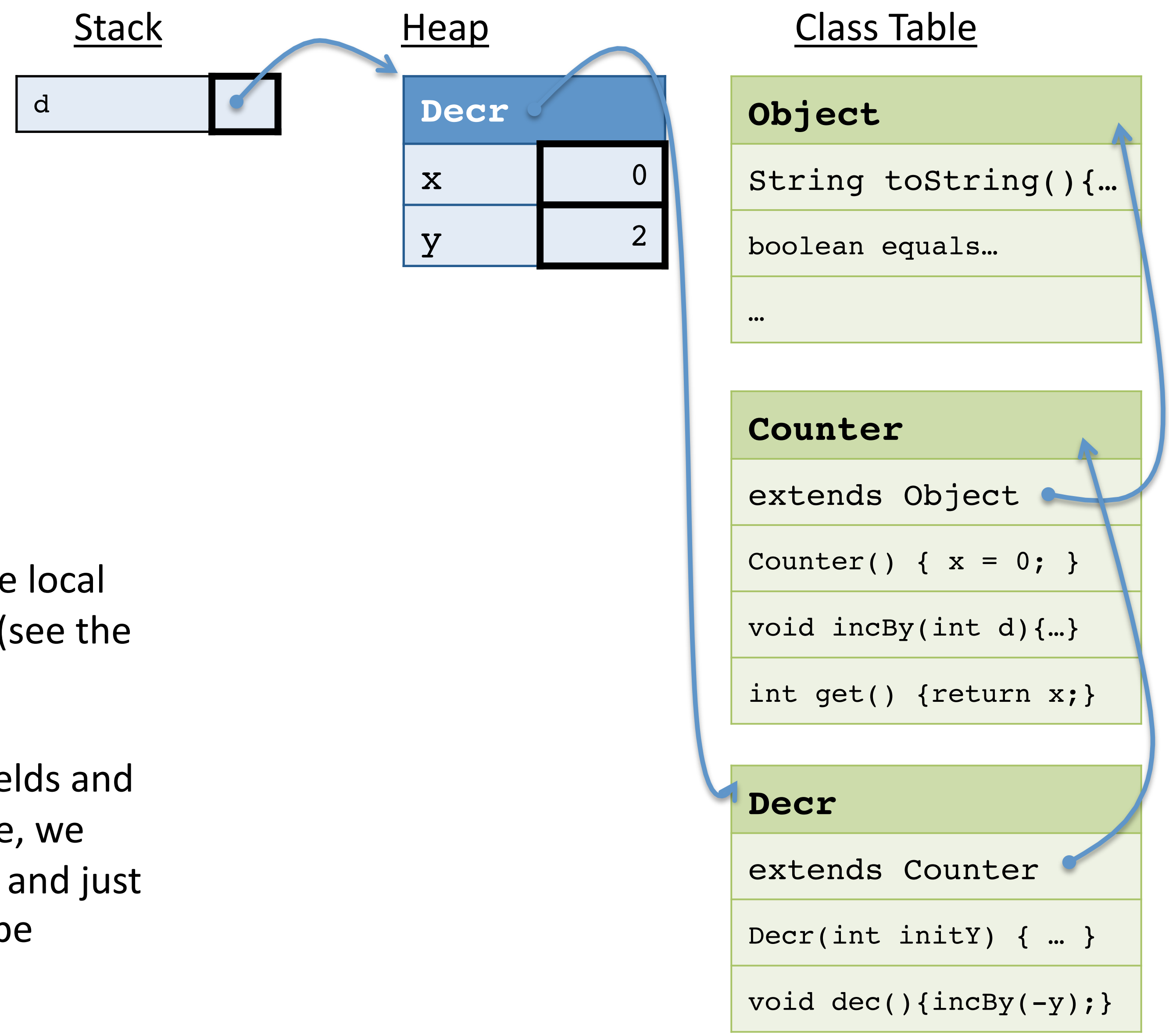


Continue executing the program.

Allocating a local variable

Workspace

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



Allocate a stack slot for the local variable d. It's mutable… (see the bold box in the diagram).

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

Dynamic Dispatch: Finding the Code

Workspace

```
•.dec();
int x = d.get();
```

Stack

d	•

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 dec method on the object. The code can be found by "pointer chasing".

This process is called *dynamic dispatch* – which code is run depends on the dynamic type 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);
```

Stack

d	
_; int x = d.get();	
this	

Heap

Decr	
x	0
y	2

Class Table

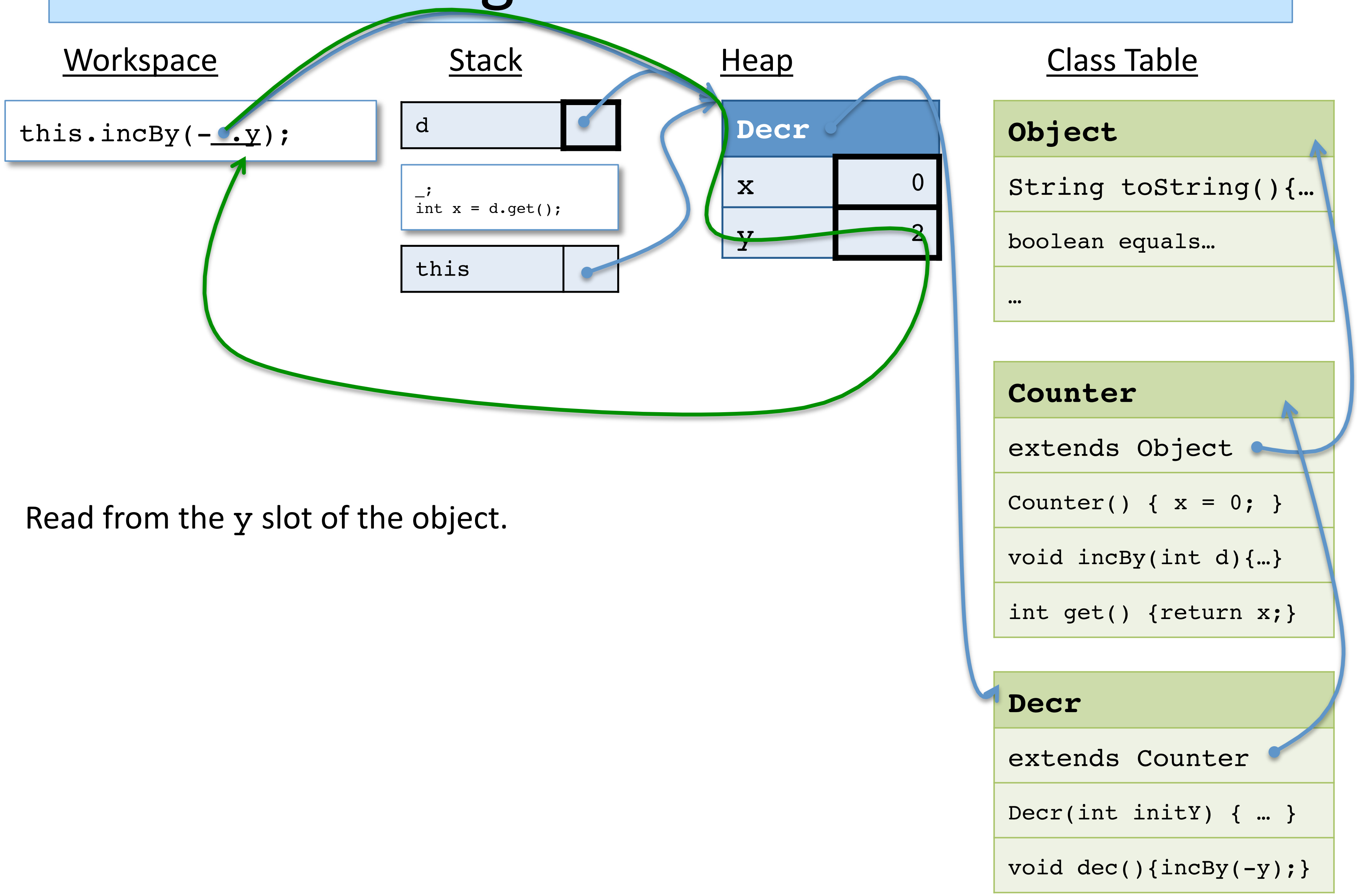
Object
String toString(){…
boolean equals…
…

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

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

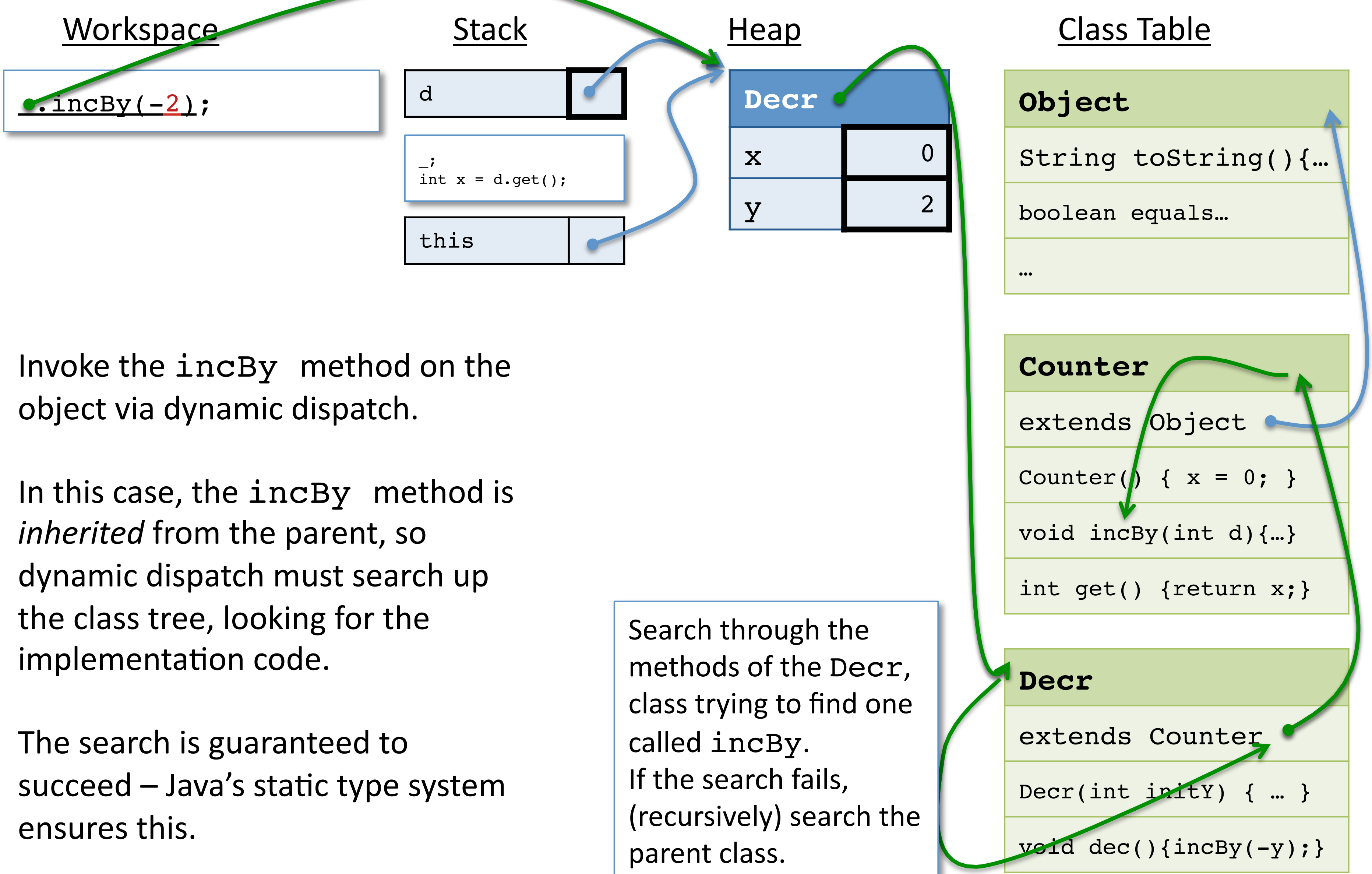
Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

Reading A Field’s Contents



Read from the y slot of the object.

Dynamic Dispatch, Again



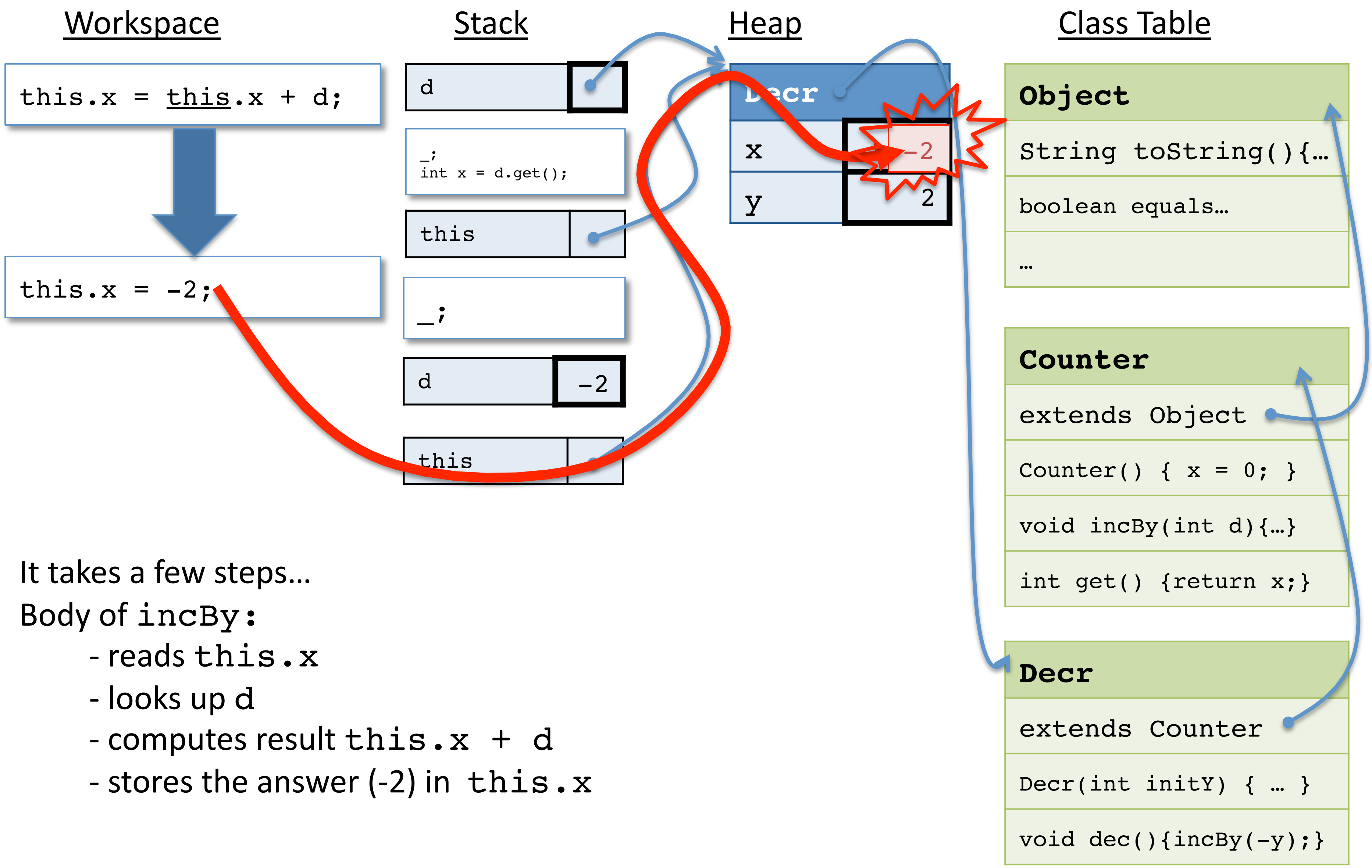
Invoke the `incBy` method on the object via dynamic dispatch.

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 trying to find one called `incBy`.
If the search fails, (recursively) search the parent class.

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…

Workspace

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

Stack

d	

Heap

Decr	
x	-2
y	2

Class Table

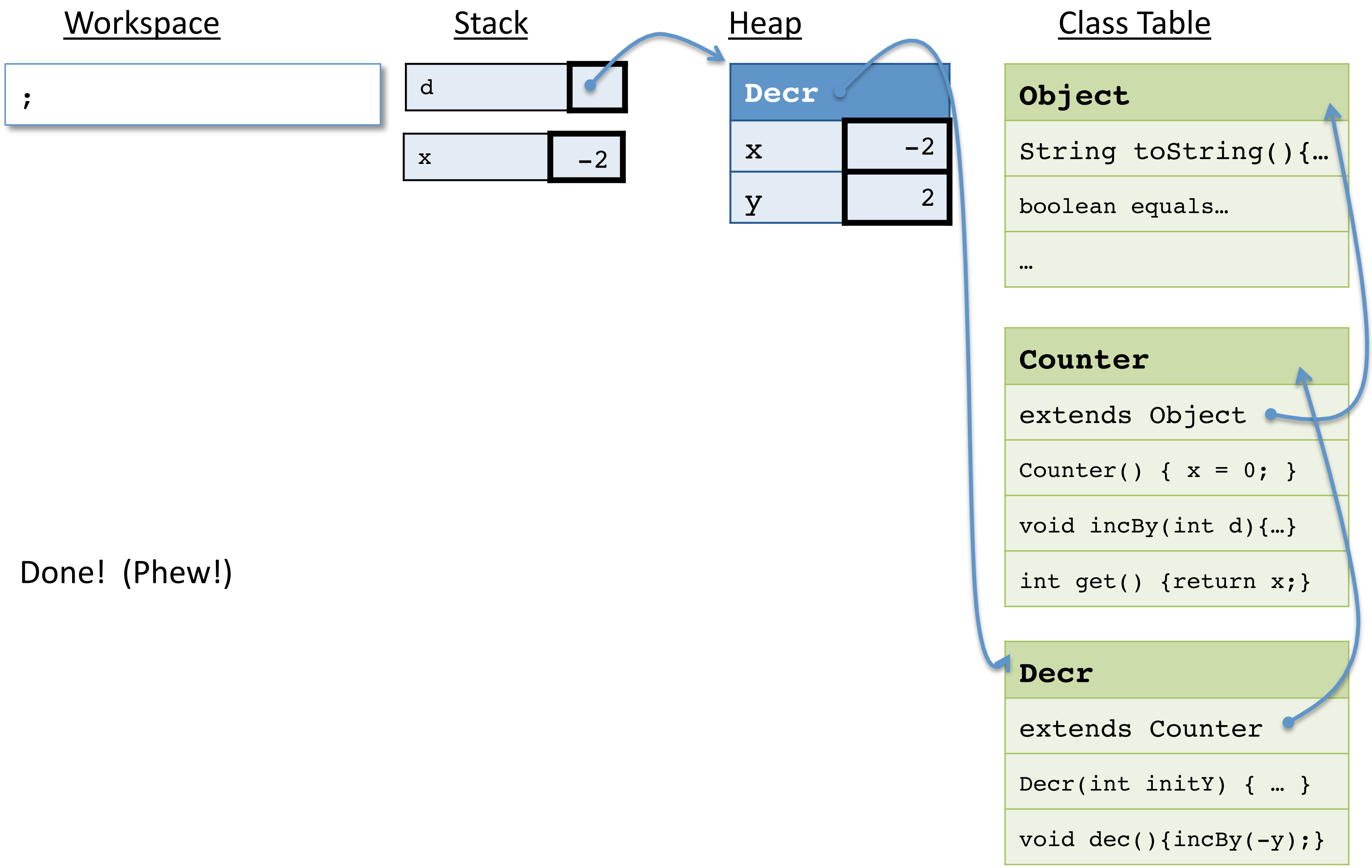
Object
String toString(){…
boolean equals…
…

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

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

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

After yet a few more steps…



Done! (Phew!)

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, which is just a pointer to a class, 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 is called *dynamic dispatch*.

- 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.