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 was used to create it (at run time)

```
x = new Point(2,3)
```

- 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

Dynamic Dispatch

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

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

Workspace

…

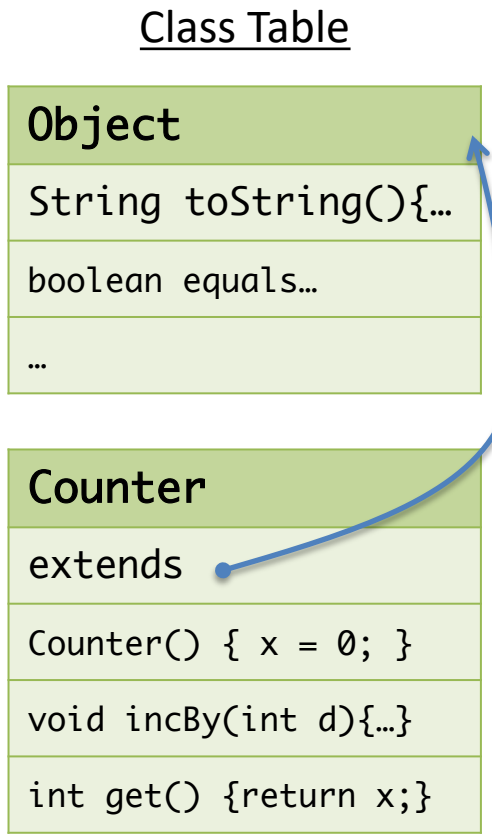
Stack

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; }
}
```



The class table contains:

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

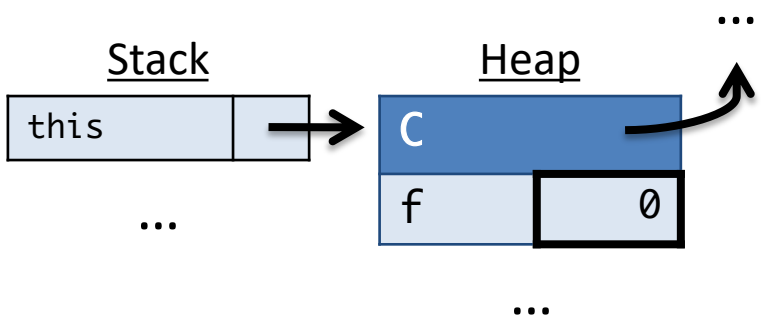
this

- 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; }
}


// … somewhere in main:
Counter d = new Counter(2);
d.incBy(2);
int x = d.get();
System.out.println(d);
```

…with Explicit this

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


// … somewhere in main:
Counter d = new Counter(2);
d.incBy(2);
int x = d.get();
System.out.println(d.toString());
```

Constructing an Object

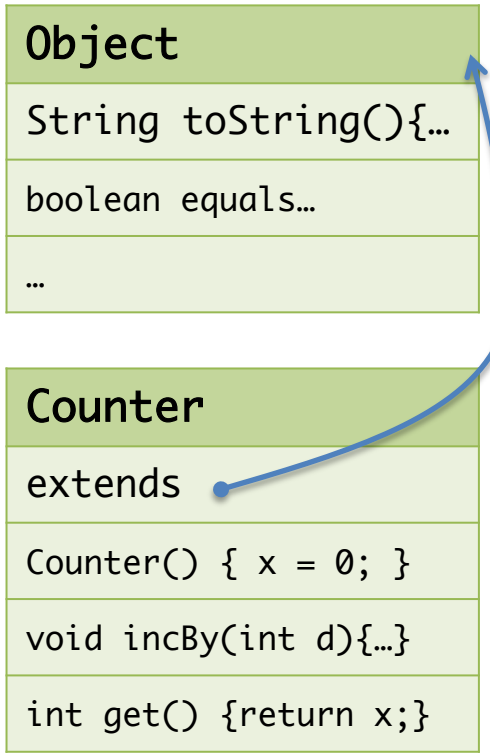
Workspace

```
Counter d = new Counter(2);
d.incBy(2);
int x = d.get();
System.out.println(d);
```

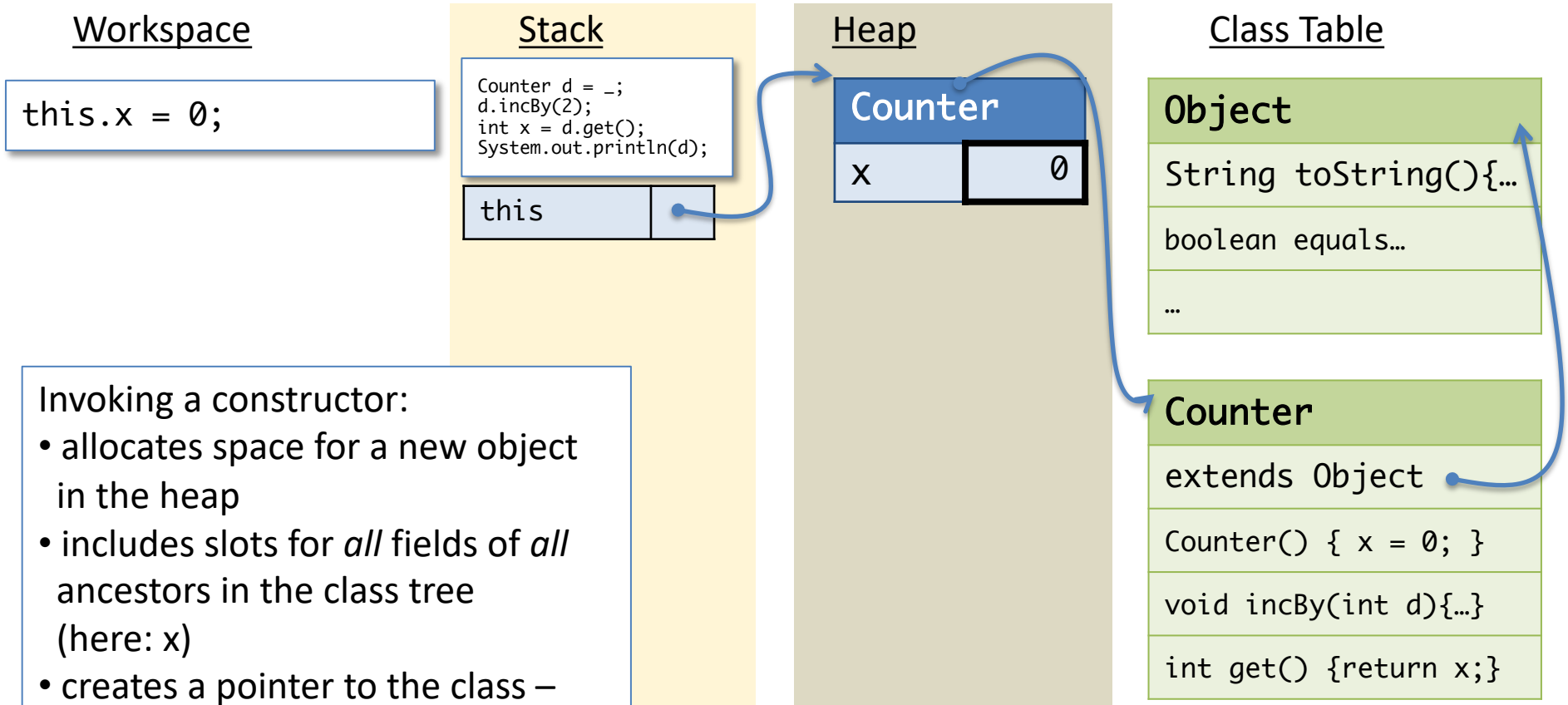
Stack

Heap

Class Table



Allocating Space on the Heap



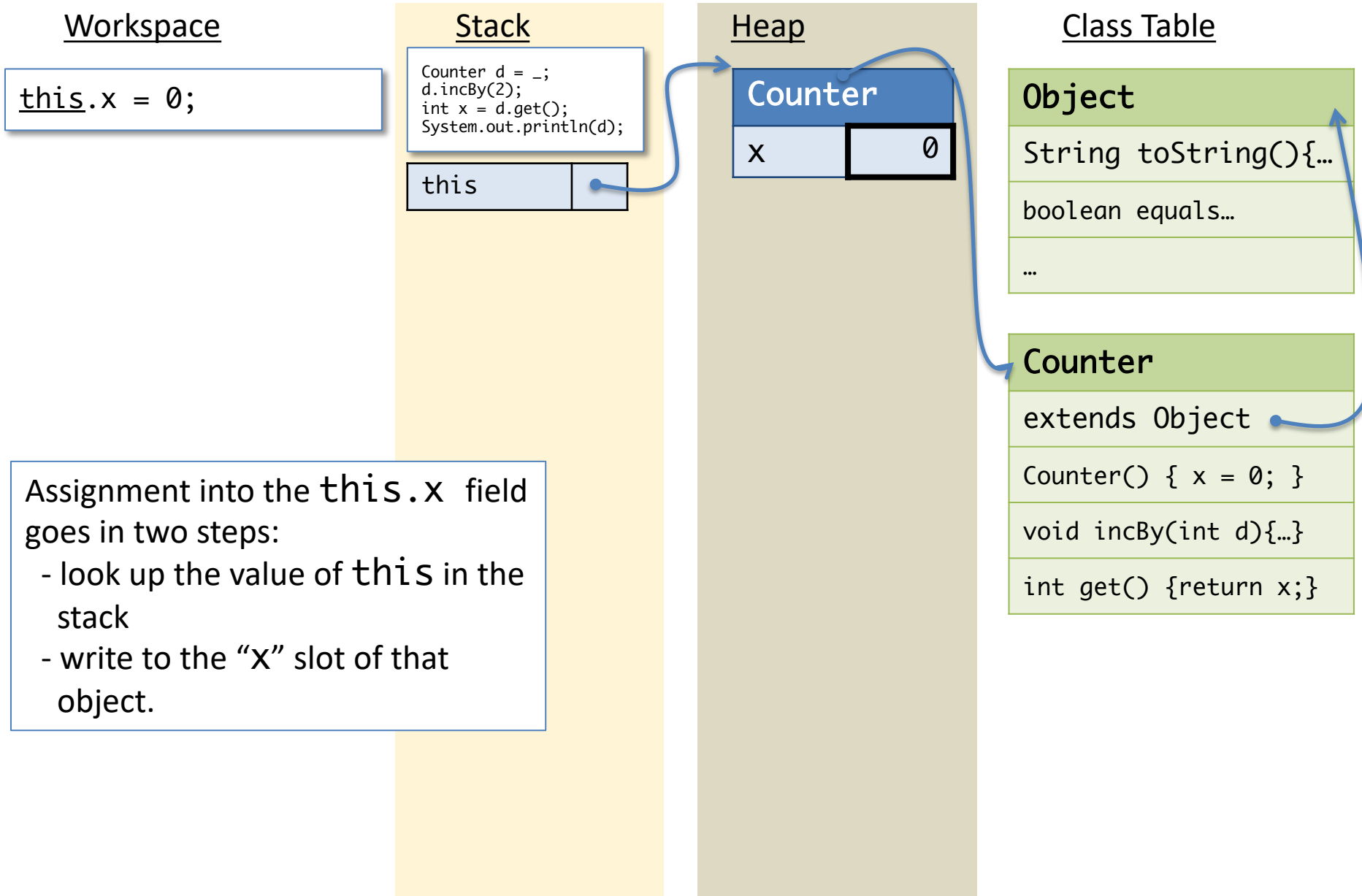
Invoking a constructor:

- allocates space for a new object in the heap
- includes slots for *all* fields of *all* ancestors in the class tree (here: x)
- 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

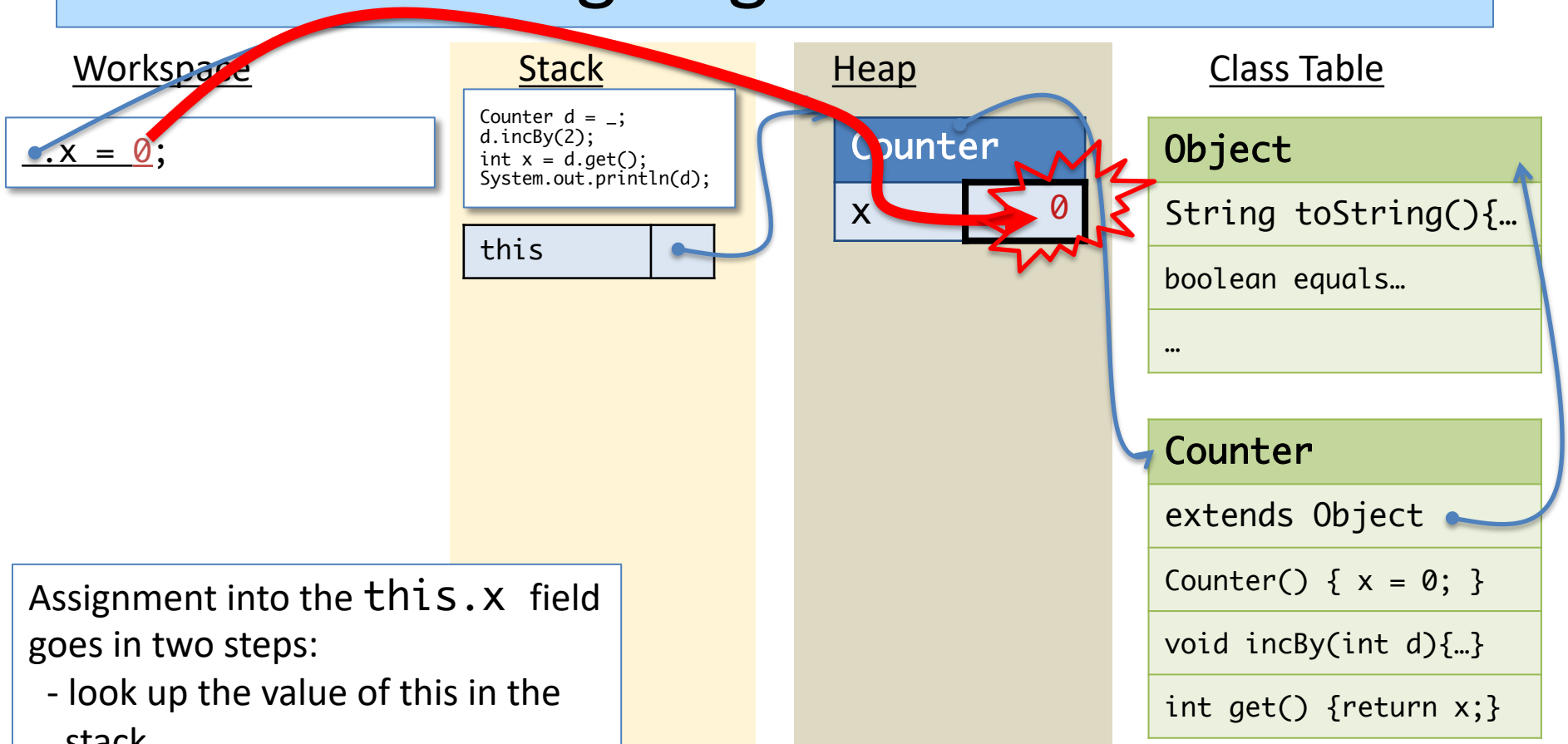
Assigning to a Field



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.

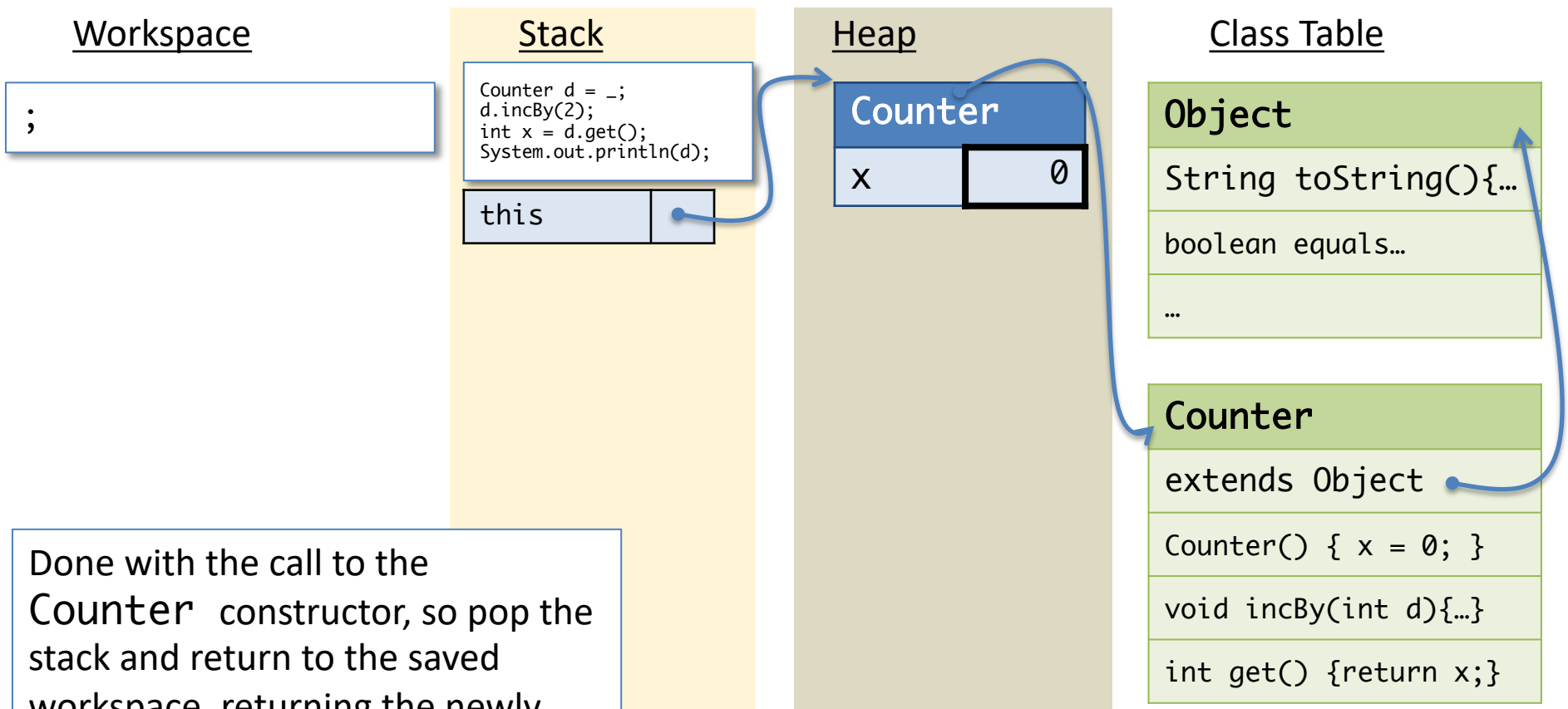
Assigning to a Field



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

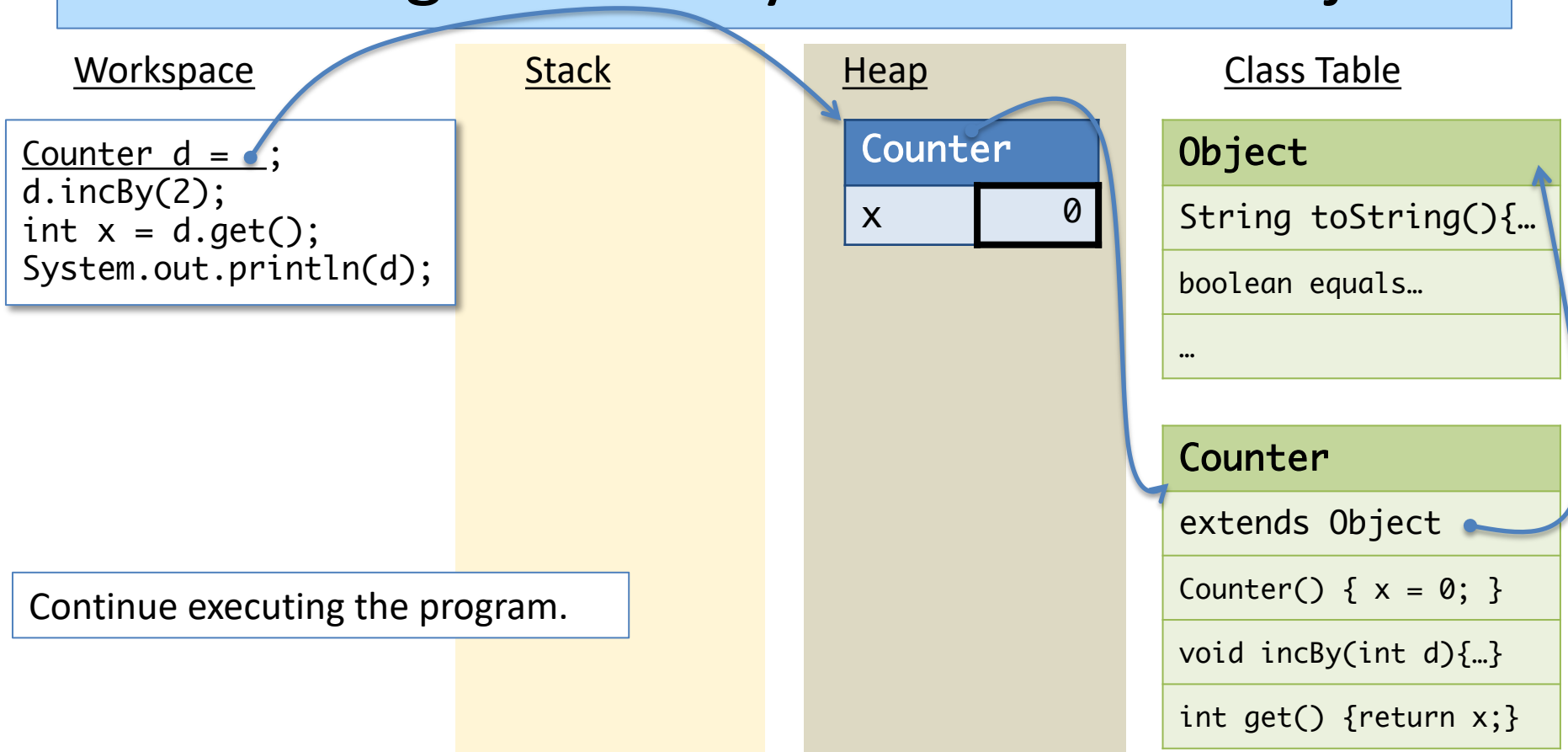
- look up the value of this in the stack
- write to the "x" slot of that object.

Done with the call



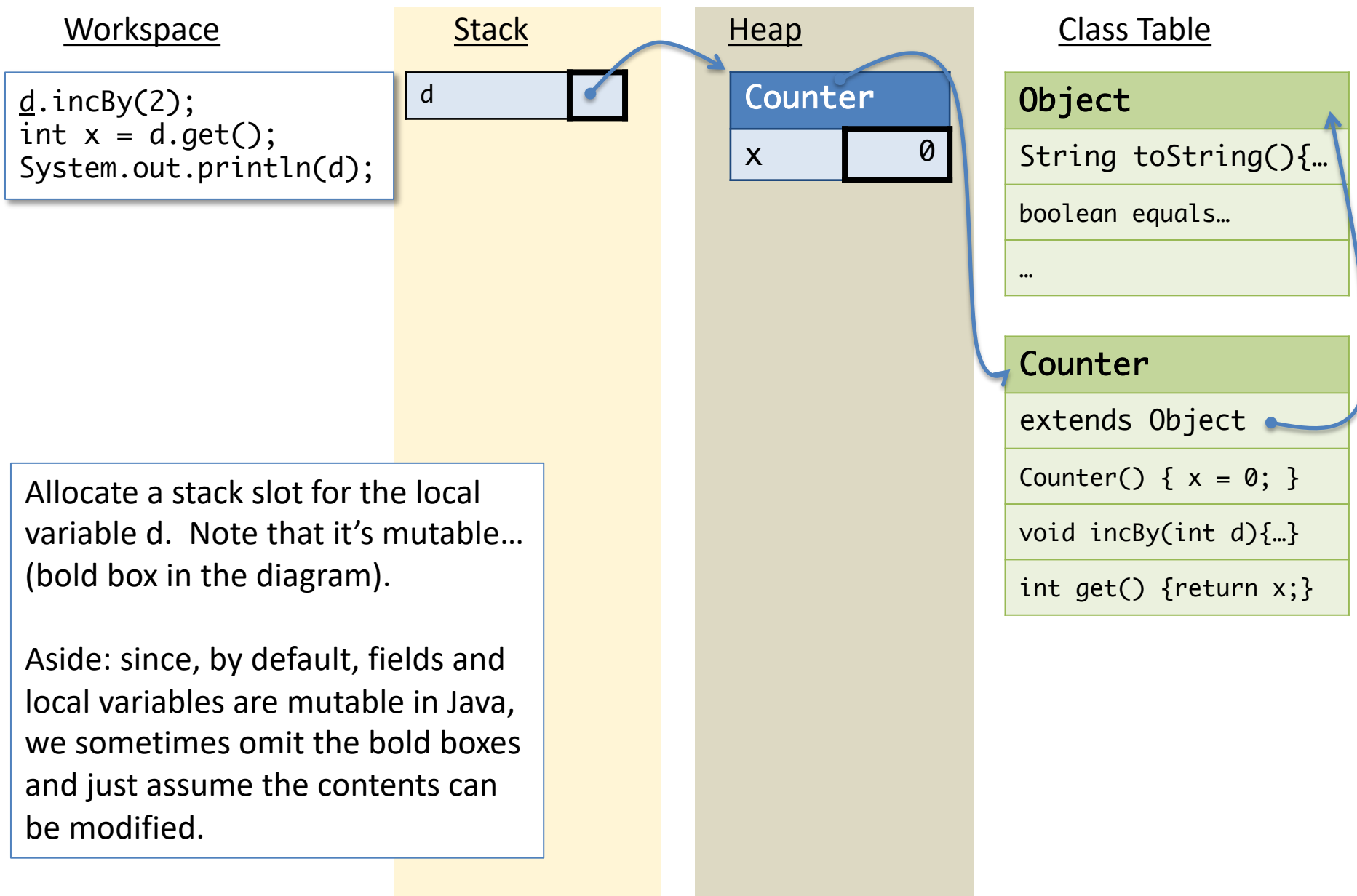
Done with the call to the `Counter` constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the `this` pointer).

Returning the Newly Constructed Object



Continue executing the program.

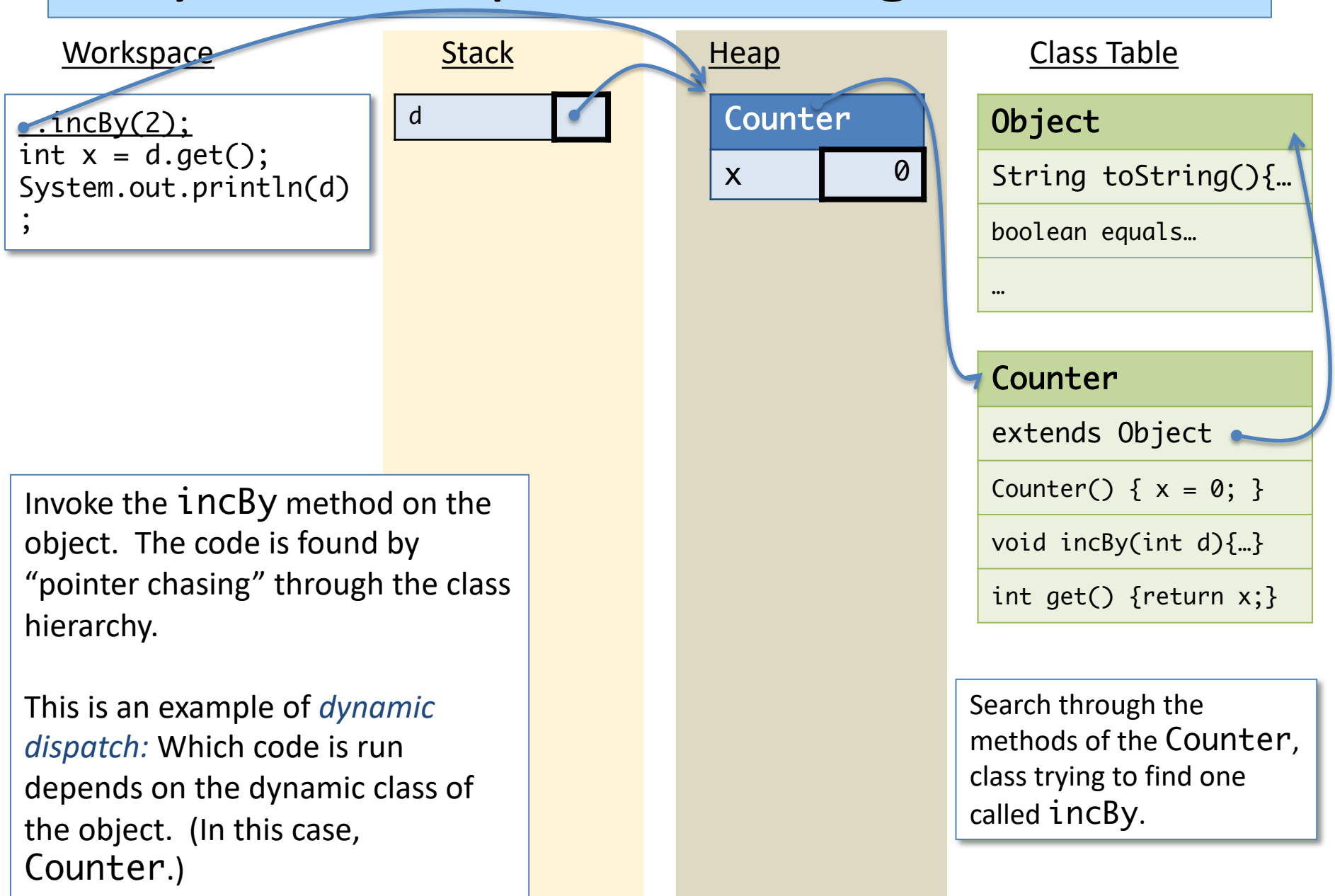
Allocating a local variable



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

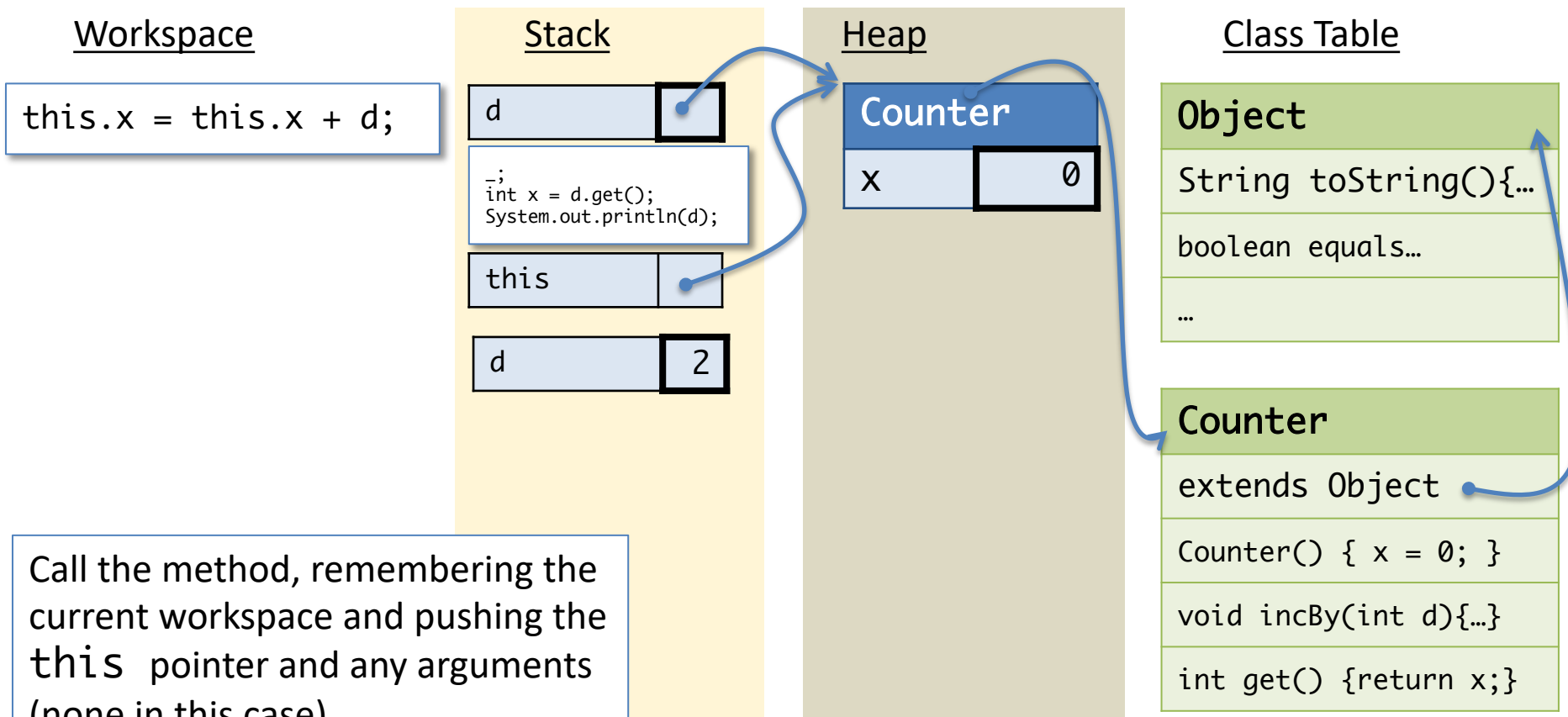


Invoke the `incBy` method on the object. The code is found by "pointer chasing" through the class hierarchy.

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

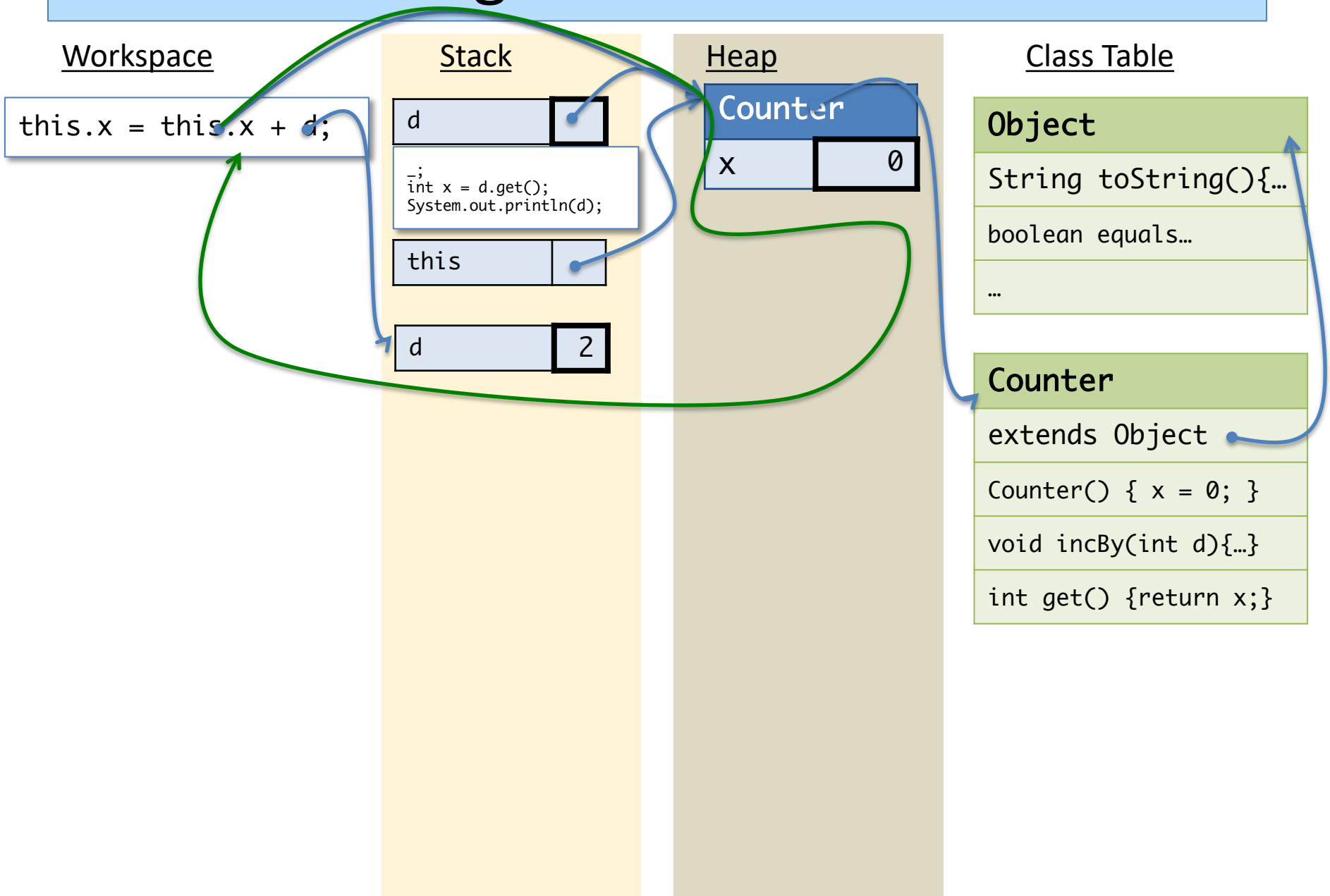
Search through the methods of the `Counter`, class trying to find one called `incBy`.

Dynamic Dispatch: Finding the Code

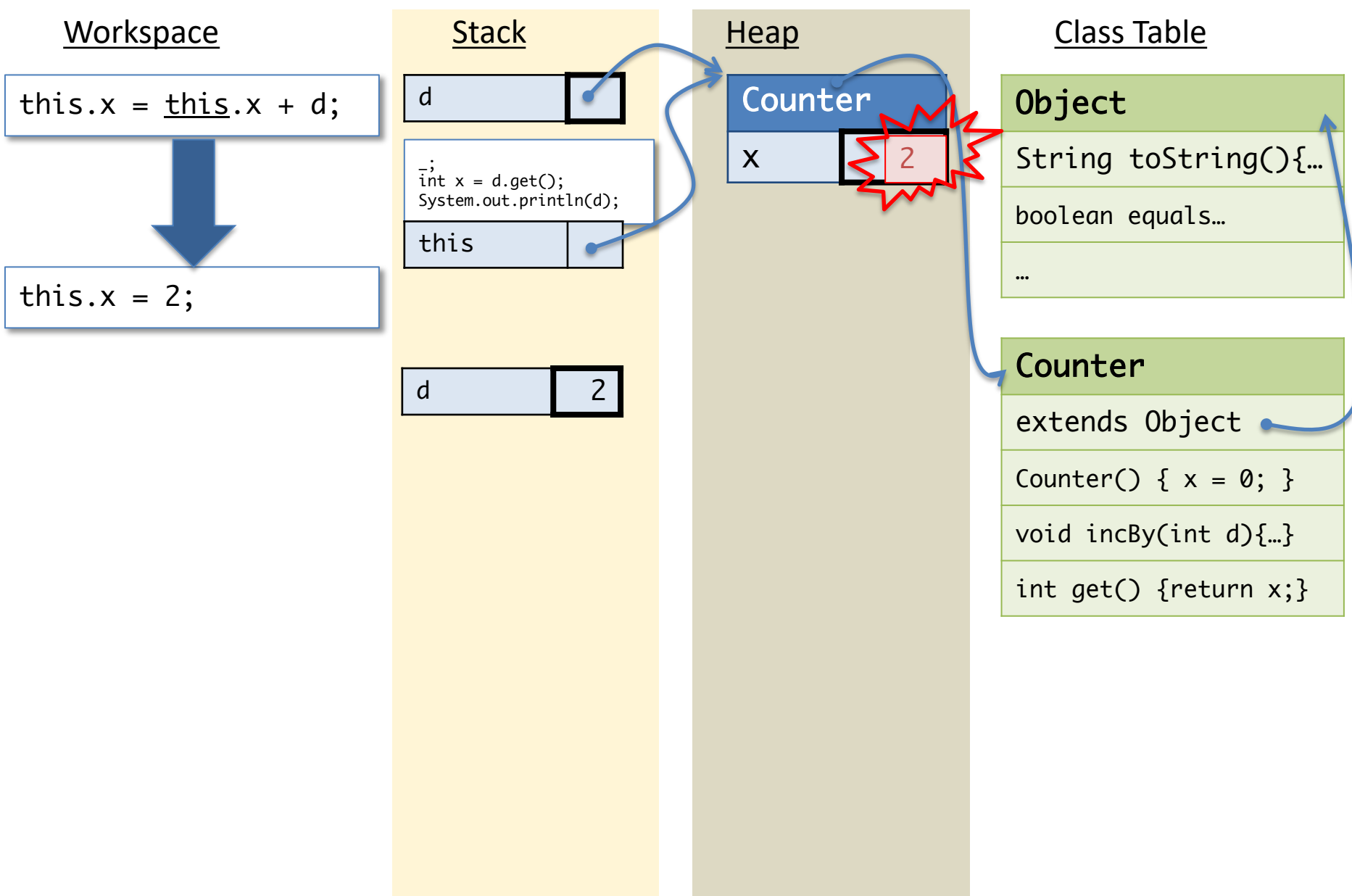


Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

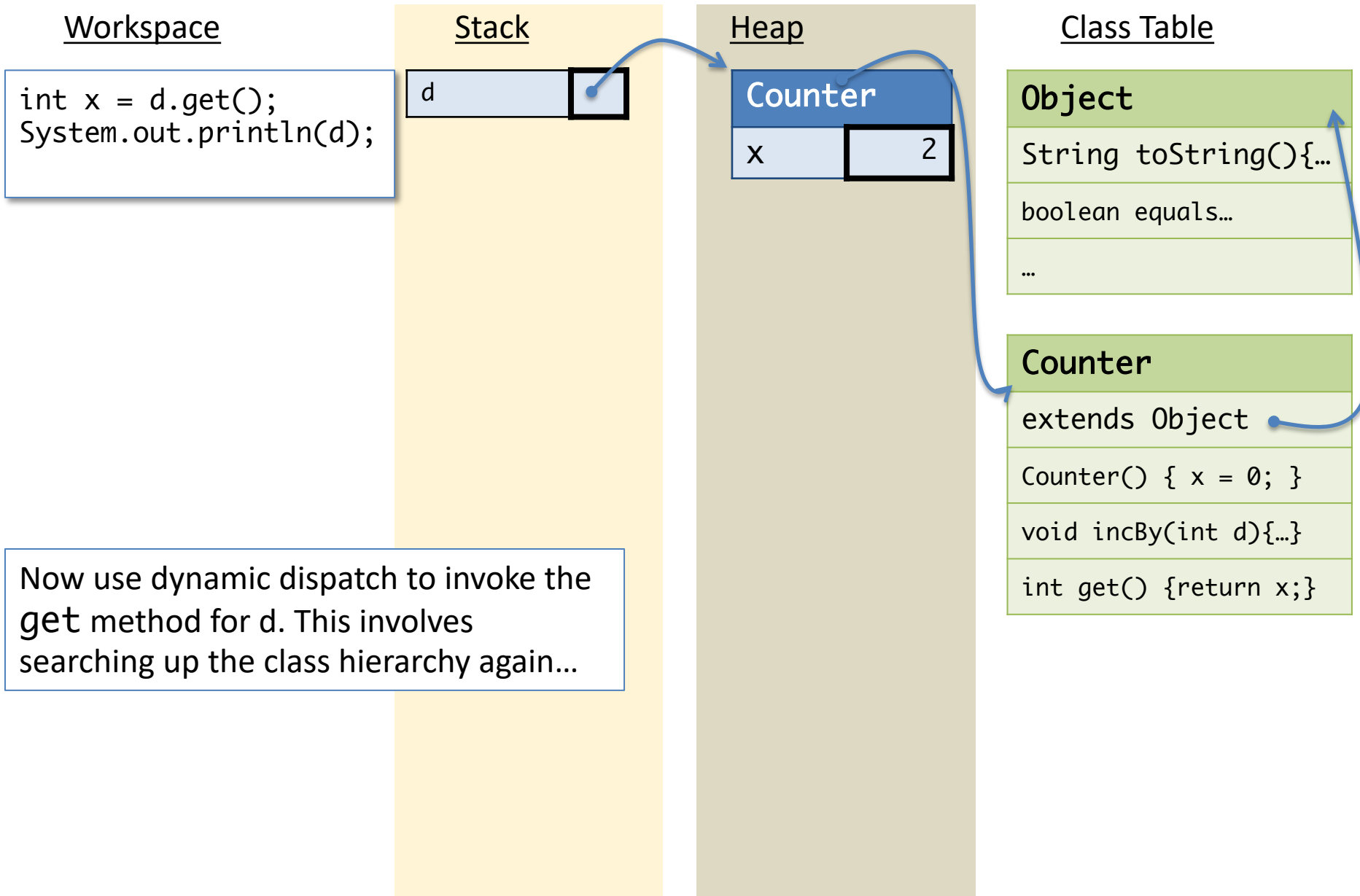
Reading a Field's Contents



Running the body of `incBy`

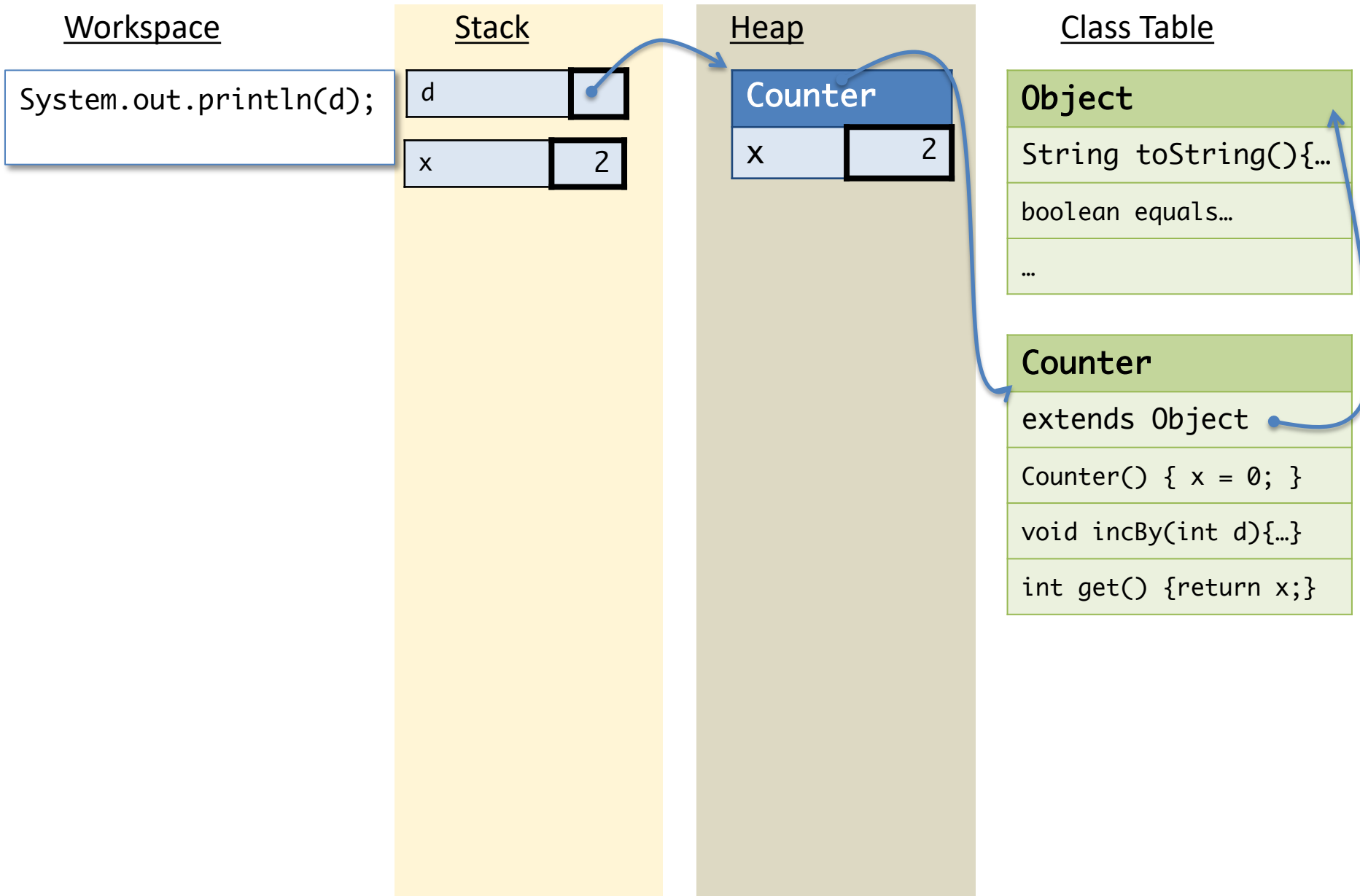


After a few more steps…

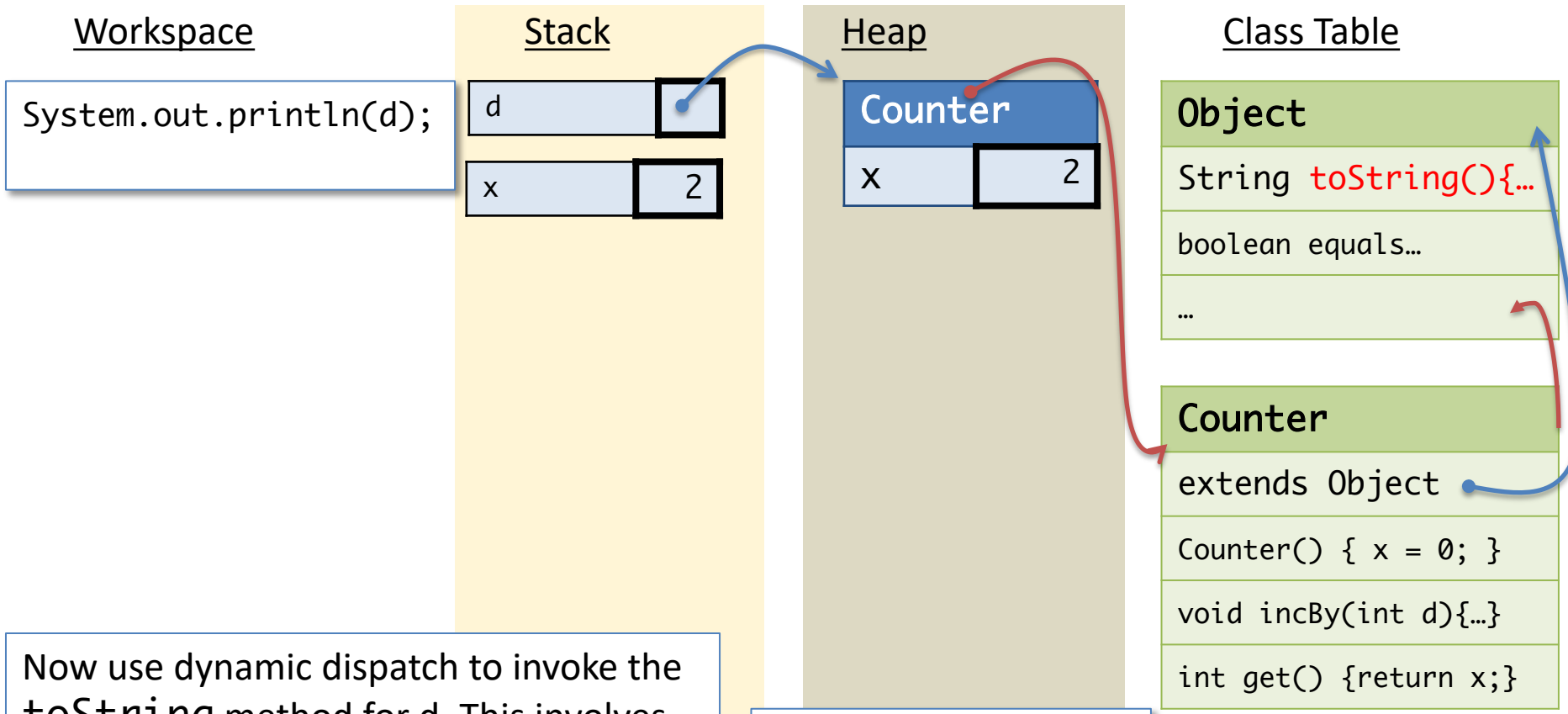


Now use dynamic dispatch to invoke the `get` method for d. This involves searching up the class hierarchy again…

After yet a few more steps…



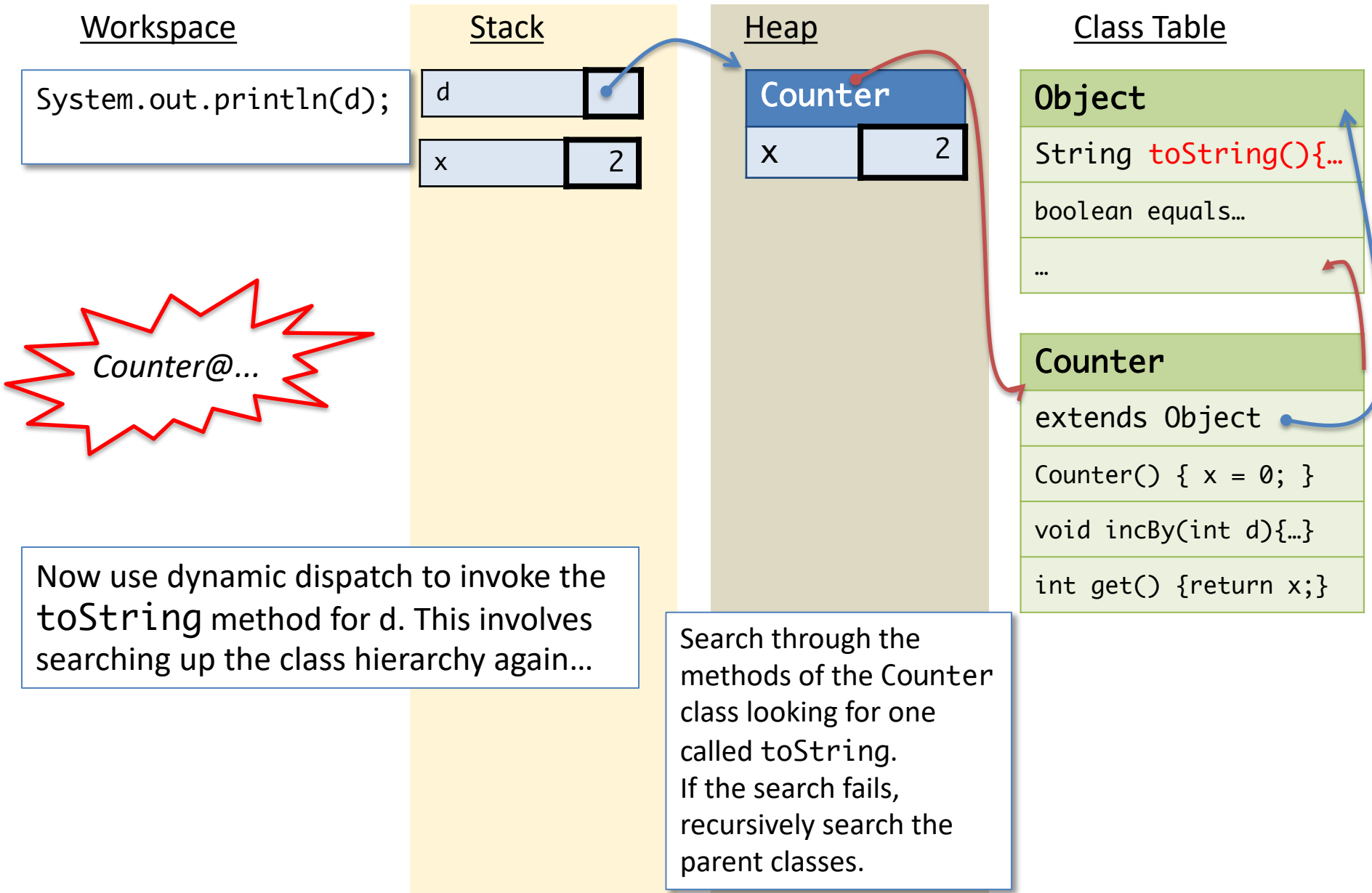
After yet a few more steps…



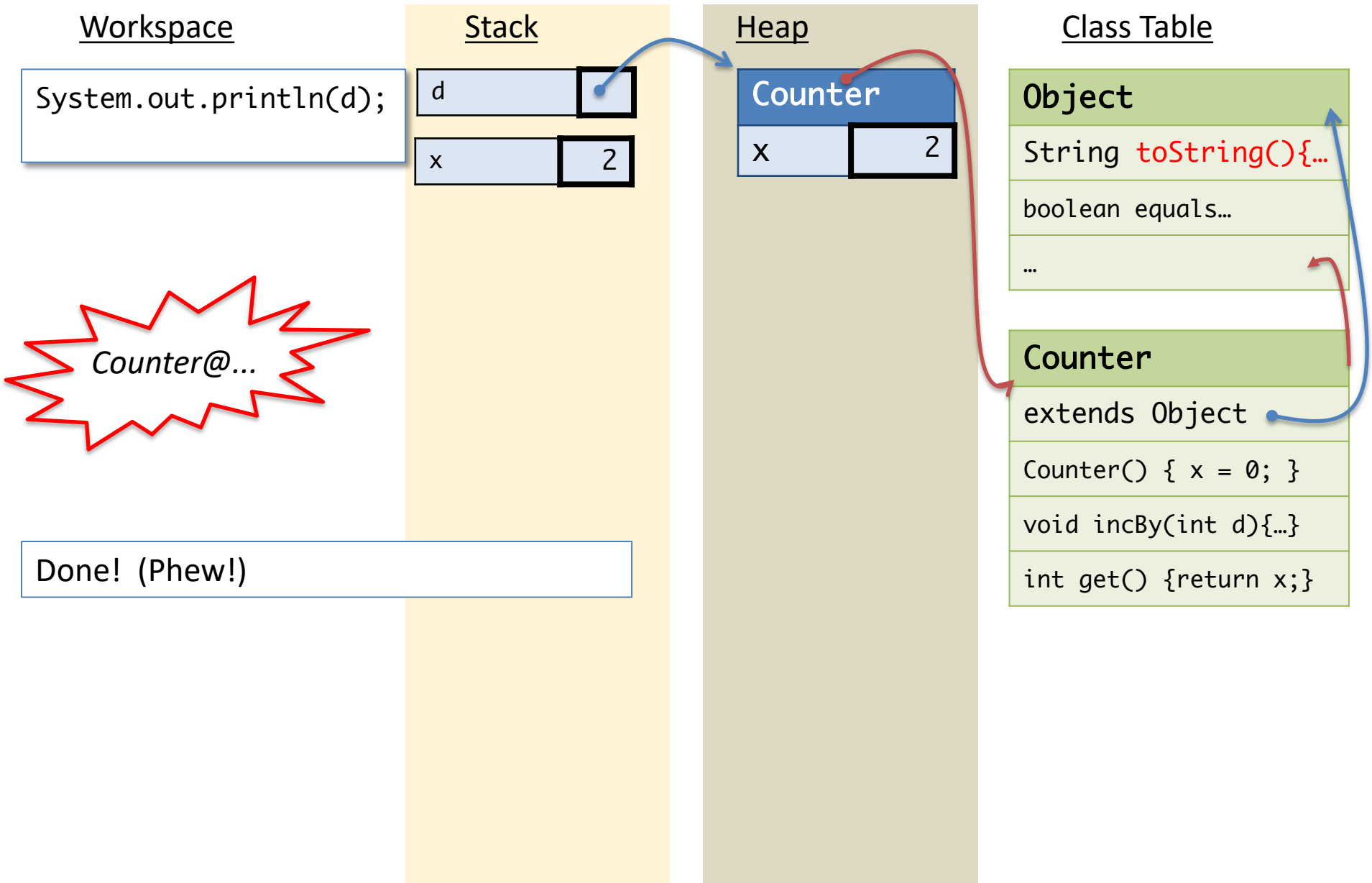
Now use dynamic dispatch to invoke the `toString` method for d. This involves searching up the class hierarchy again…

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

After yet a few more steps…



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.