Programming Languages and Techniques (CIS120)

Lecture 26

Static Types vs. Dynamic Classes

The Java ASM

Chapters 23 & 24

Quick Review:
Java Types and Interfaces

Review: Static Types

- Types stop you from using values incorrectly
  - `3 + true`
  - `(new Counter()).m()`
- All *expressions* have types
  - `3 + 4` has type `int`
  - `"A".toLowerCase()` has type `String`
- How do we know if `x.m()` is correct? or `x+3`?
  - depends on the type of `x`
- Type restrictions preserve the types of variables
  - assignment "x = 3" must be to values with compatible types
  - methods "o.m(3)" must be called with compatible arguments

HOWEVER: in Java, values can have *multiple* types....

Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Example: Interface for objects that have a position and can be moved

keyword

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

No fields, no constructors, no method bodies!

Implementing the interface

- A class that implements an interface must provide appropriate definitions for the methods specified in the interface

```
public class Point implements Displaceable {
  private int x, y;
  public Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void move(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

interfaces implemented

methods required to satisfy contract

Another implementation

```
public class Circle implements Displaceable {

   public Circle(int x, int y, int initRadius) { … }

   public int getX() { … }
   public int getY() { … }
   public void move(int dx, int dy) { … }

}
```

Objects constructed from different classes can satisfy the same interface

Implementing multiple interfaces

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

```
public class Circle implements Displaceable, Area {

   // constructor
   // implementation of Displaceable methods

   // new method
   public double getArea() {
      ...
   }
}
```

Classes can implement multiple interfaces by including *all* of the required methods

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.

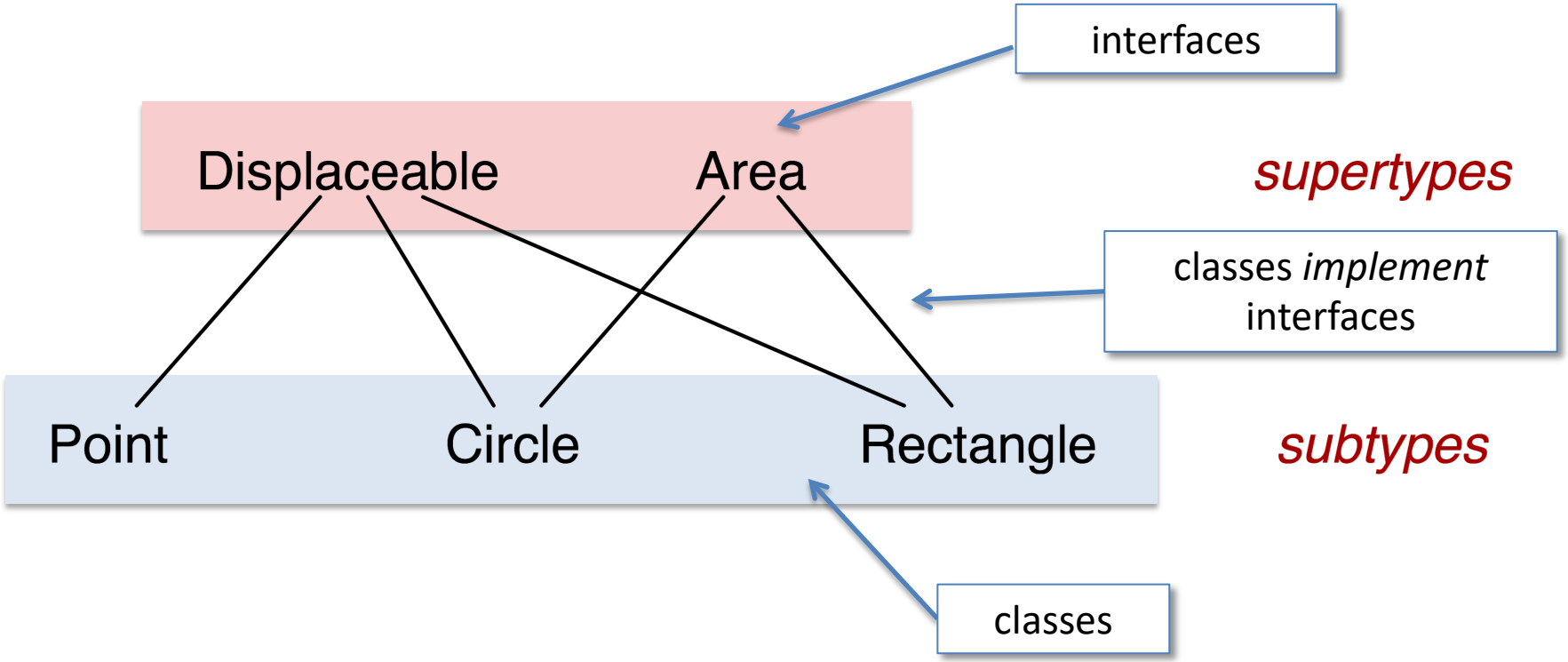
Example: A class that implements an interface declares a subtyping relationship

`Point` is a subtype of `Displaceable`

`Circle` is a subtype of `Displaceable` and `Area`

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));
```

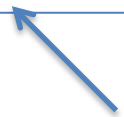
- If B is a subtype of A, it provides all of A's (public) methods
- The behavior of a nonstatic method (like move) depends on the implementation in class B

**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

Interface Extension – An interface that *extends* another interface declares a subtype

Class Extension – A class that *extends* another class declares a subtype

Interface Extension

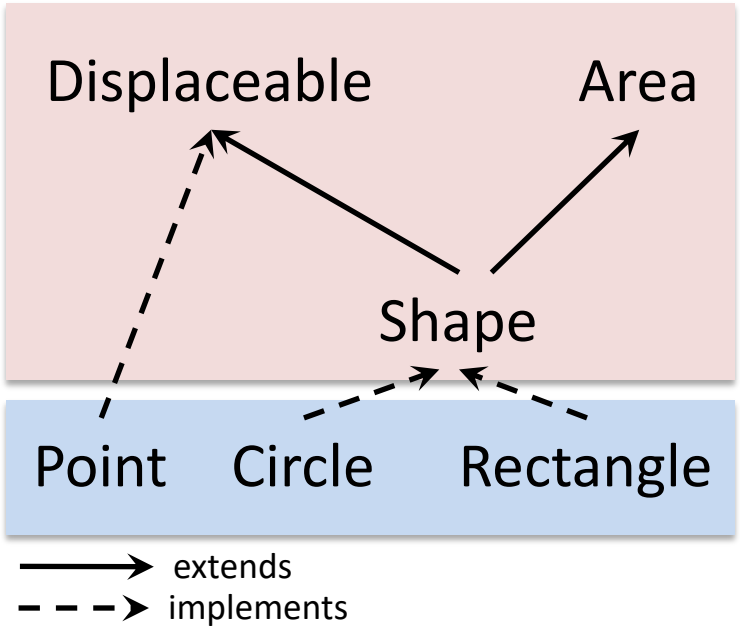
- Build richer interface hierarchies by *extending* existing interfaces.

```
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.

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; 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 interface hierarchy has no cycles.

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).

- Design Tip: Class extension should *never* be used when "is a" does not relate the subtype to the supertype.

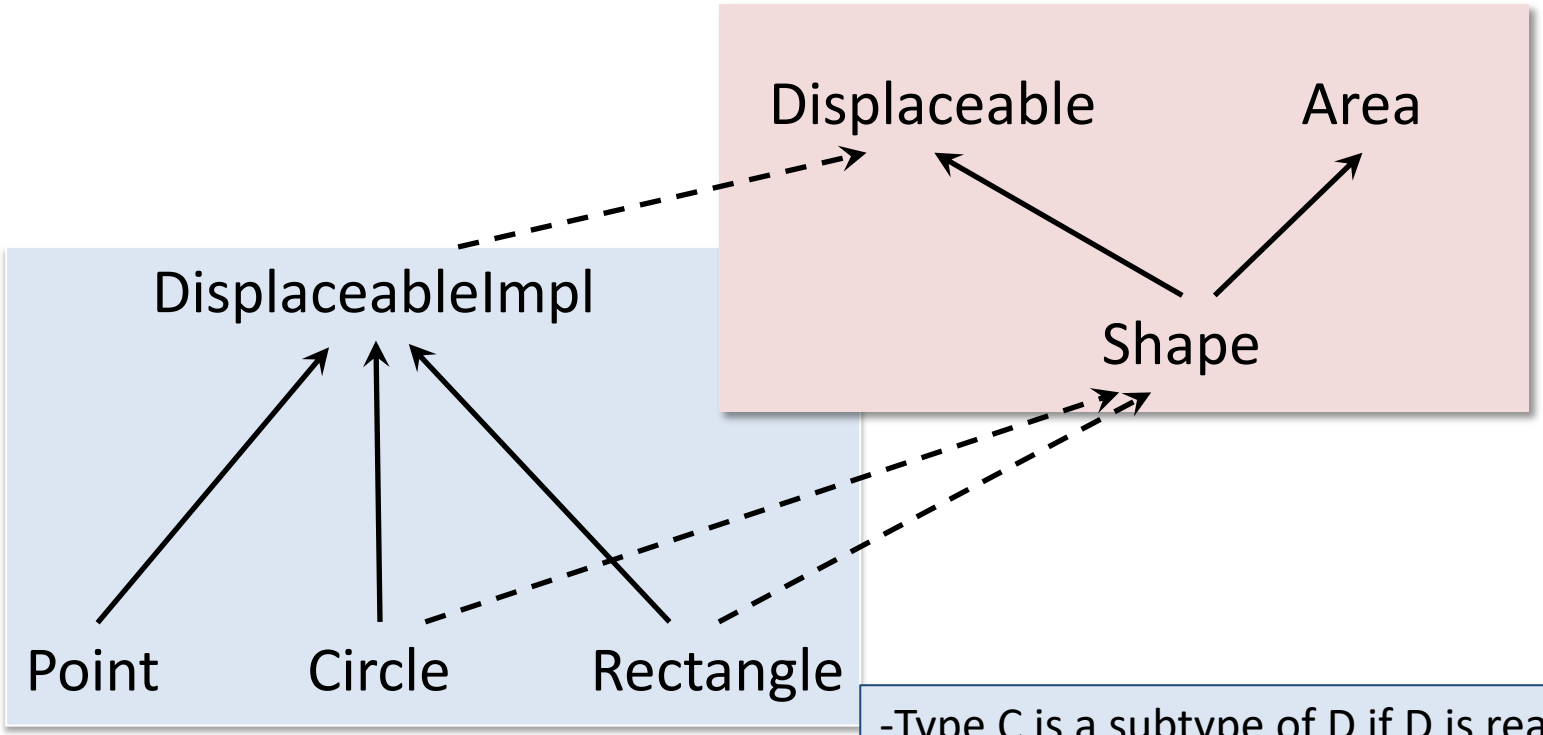
Simple Inheritance

- In *simple inheritance*, a subclass only *adds* new fields or methods.

- Use simple inheritance to *share common code* among related classes.

- **Example**: Suppose `Point`, `Circle`, and `Rectangle` have *identical* code for `getX()`, `getY()`, and `move()` methods when implementing `Displaceable`. Let's put this code in one place.

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

Gotcha: 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 drop the call to `super`)

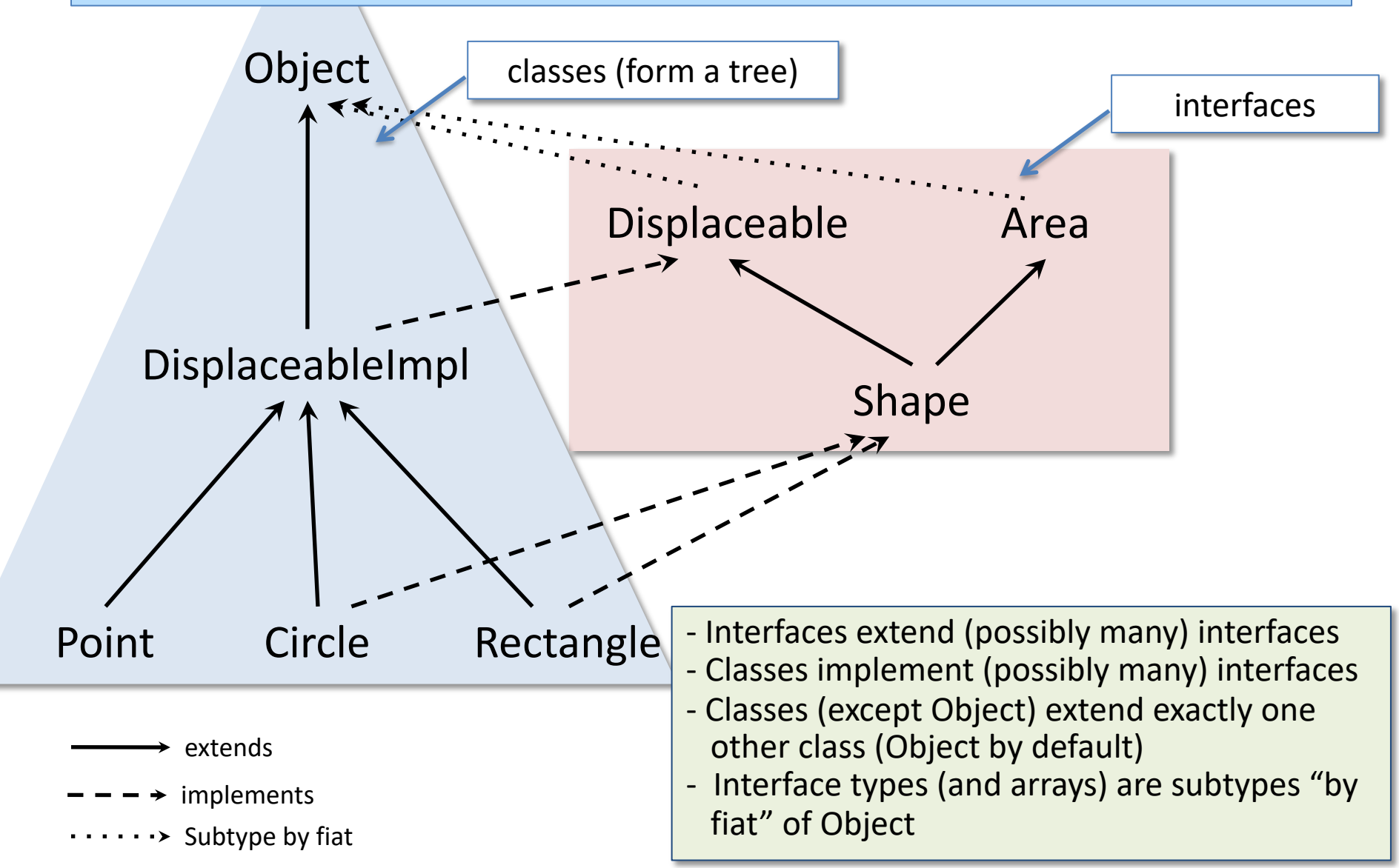
```
public Circle(Point pt, int radius) {
     super(pt.getX(),pt.getY());
     this.radius = radius;
   }
```

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
  - This 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)
- These features are tricky to use properly, and the need for them arises only in special cases
  - Designing complex, reusable libraries
  - 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 in a few special cases

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)
```

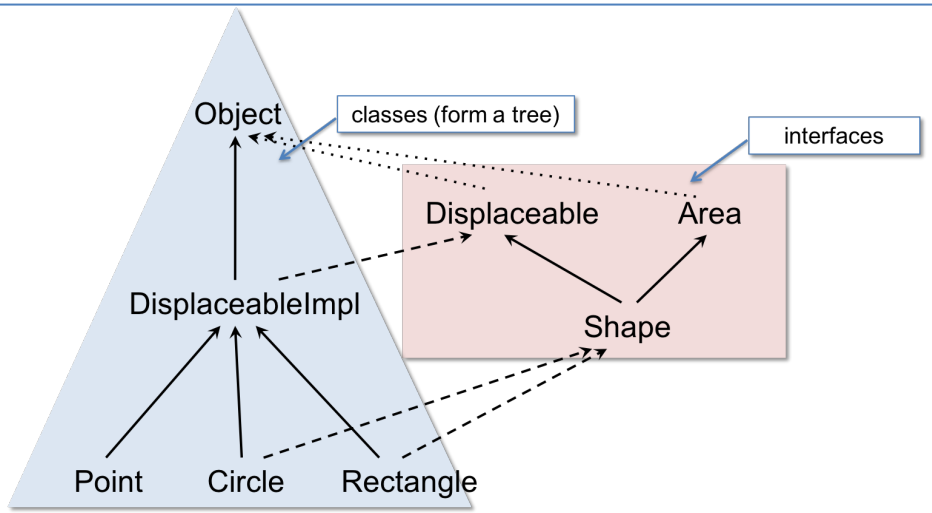
- In OCaml, we only had static types
- In Java, we also have dynamic classes because of objects
  - The dynamic class will always be a *subtype* of its static type
  - The dynamic class determines what methods are run

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



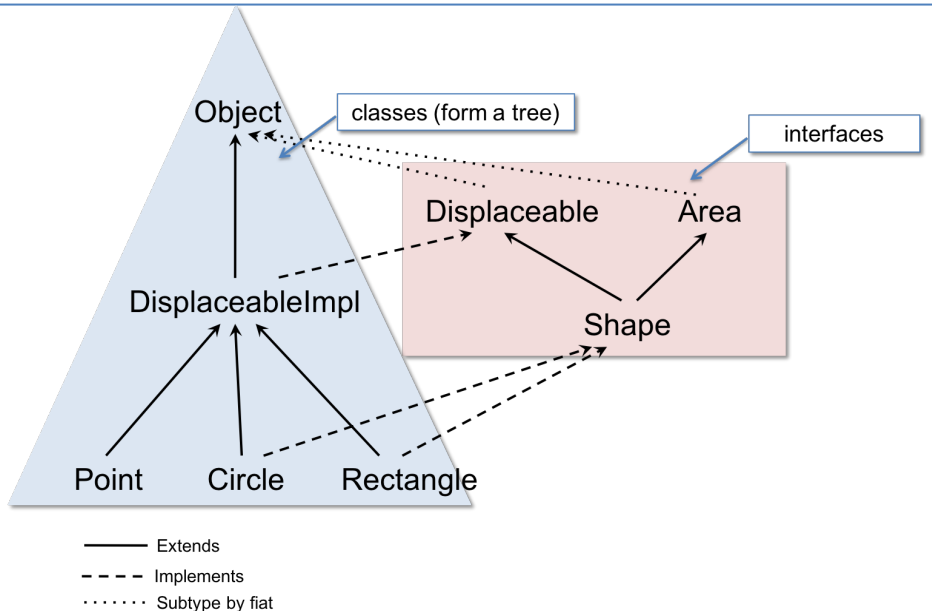
Area

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 dynamic class of a1 when execution reaches A?

1. Area
2. Circle
3. None of the above
4. Not well typed



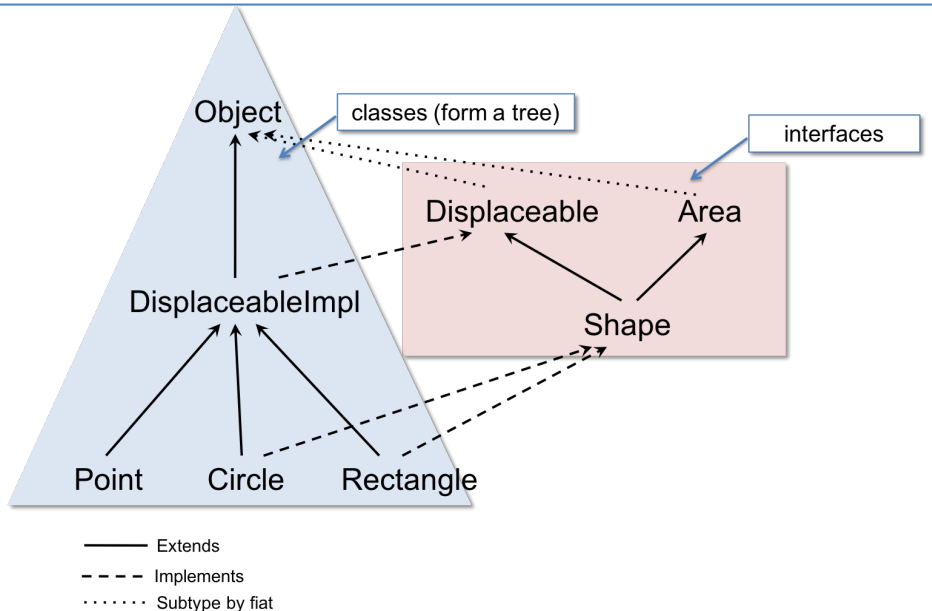
Circle

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 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’?

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

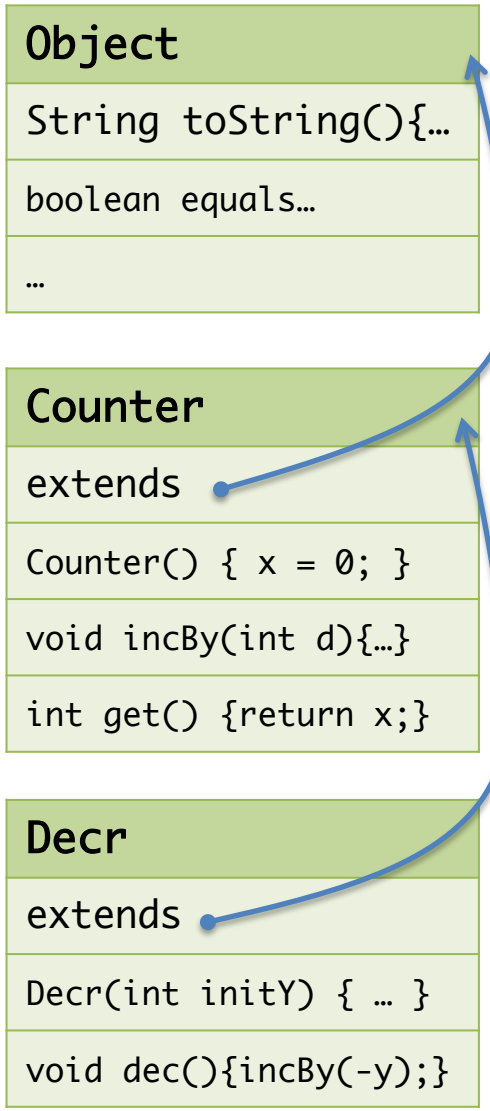
Heap

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; }
}

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:

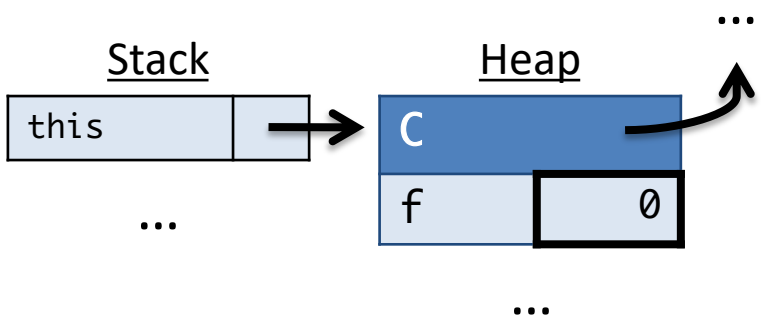
- **the code for each method**,
- **references to each class's parent**, and
- the class's static members.

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 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

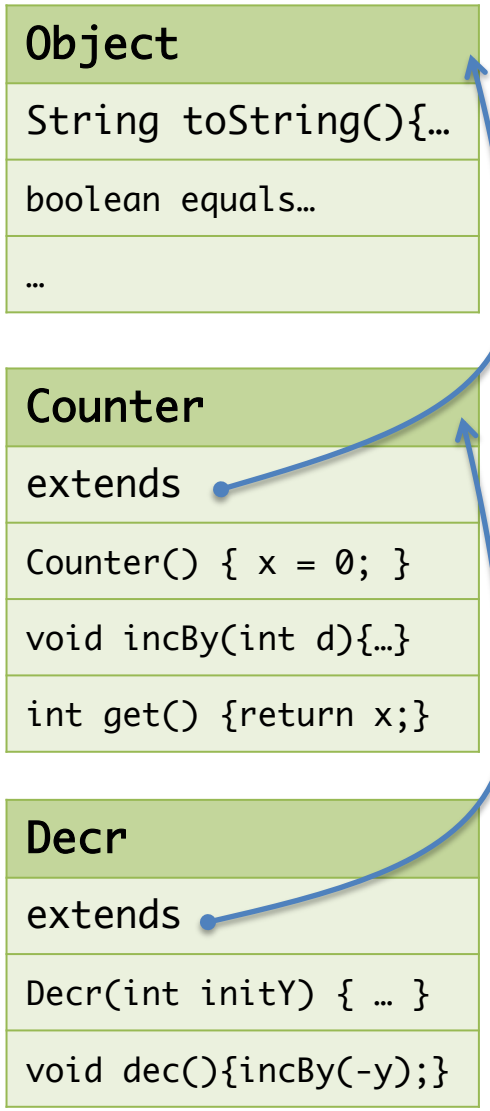
Workspace

```
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

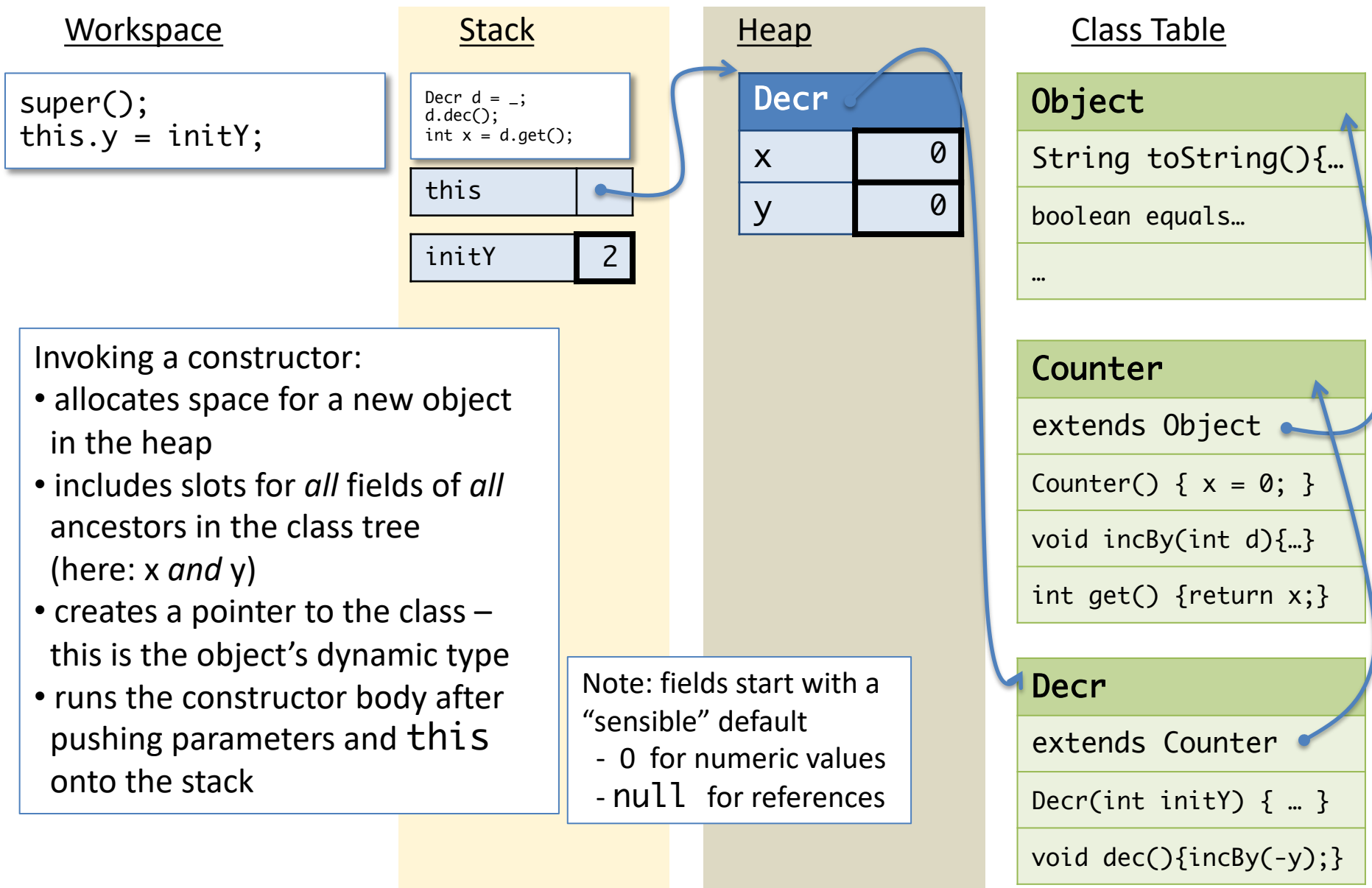
Stack

Heap

Class Table



Allocating Space on the Heap



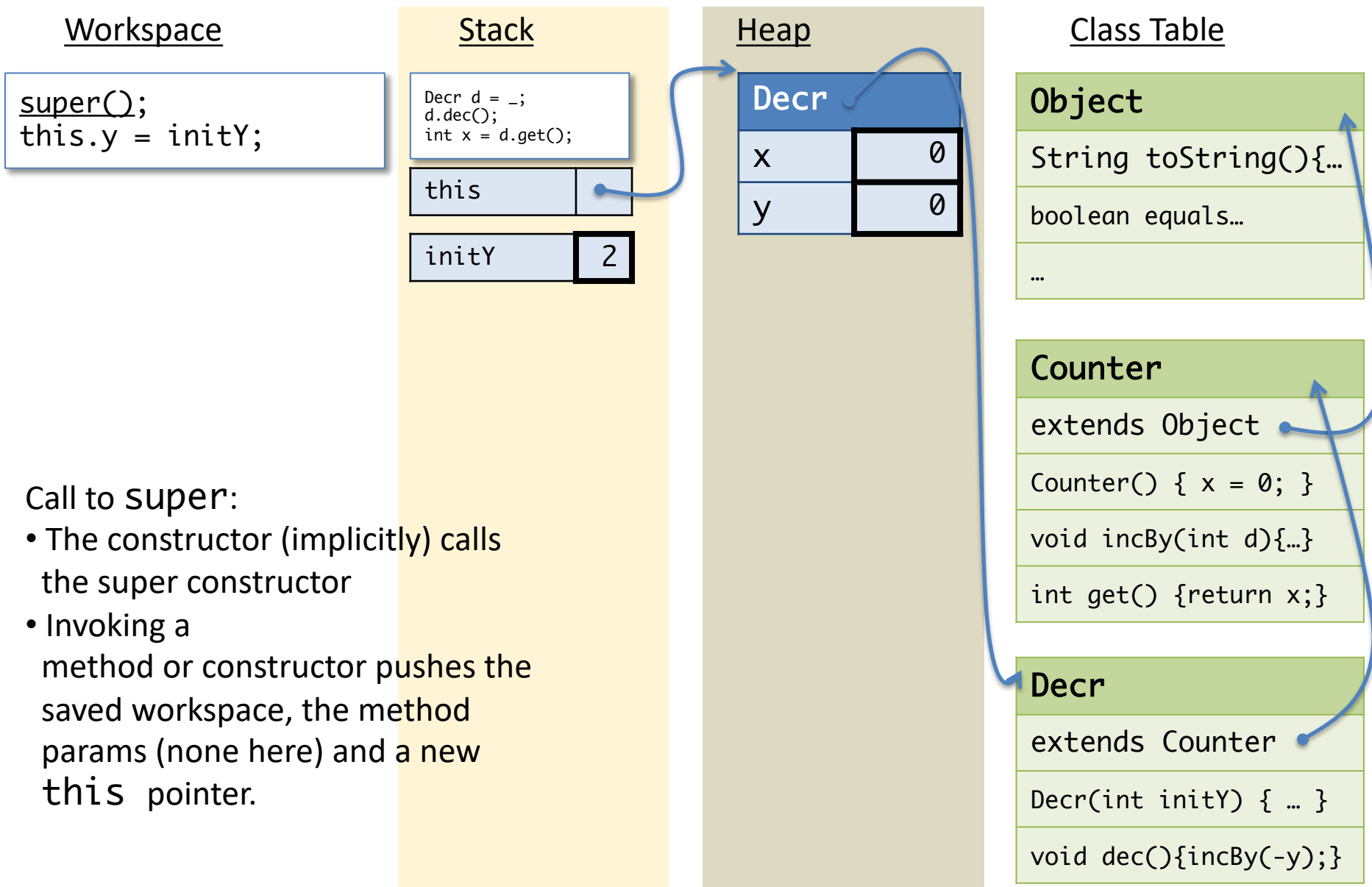
Invoking a constructor:

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

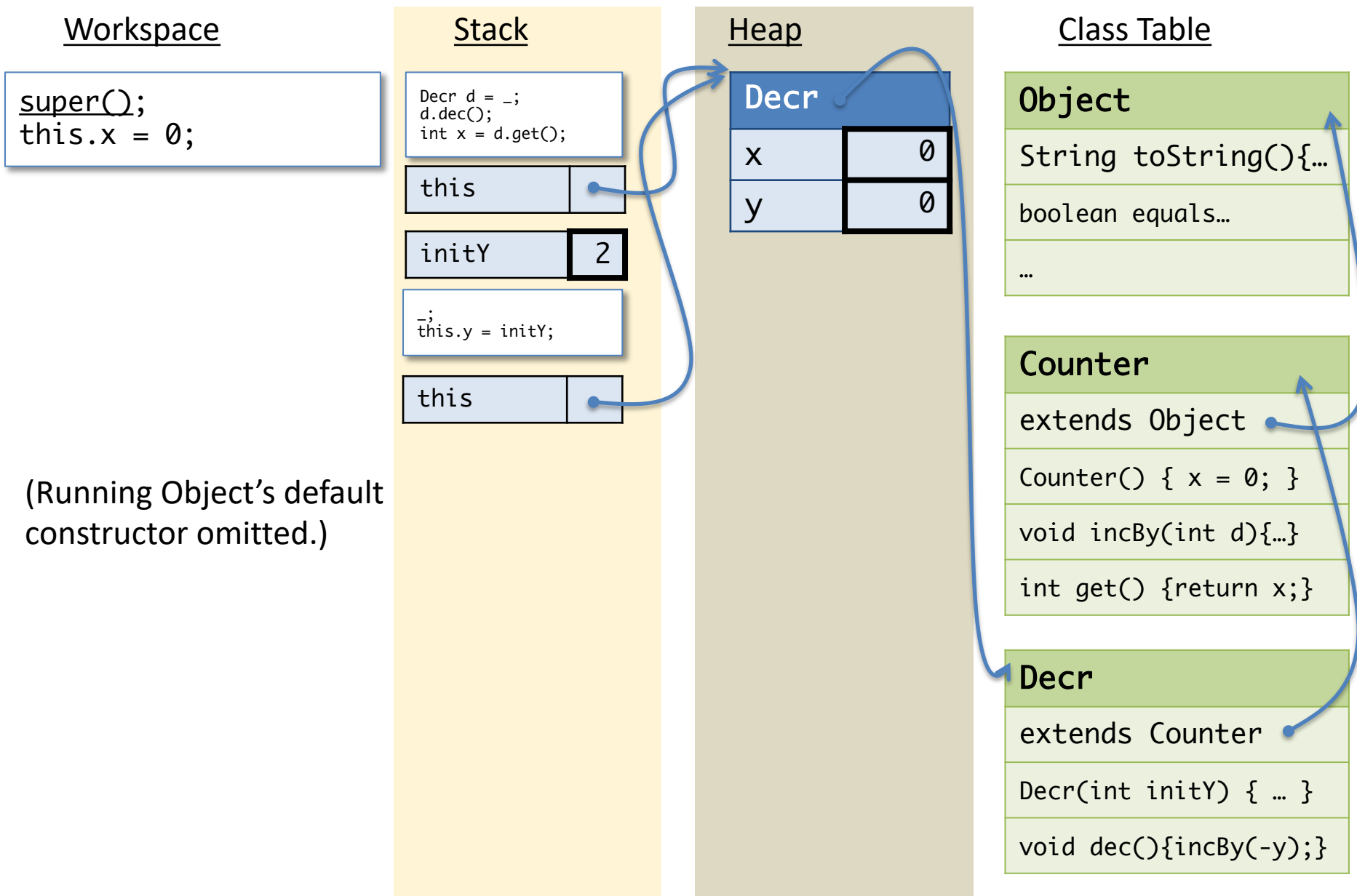
Calling super



Call to super:

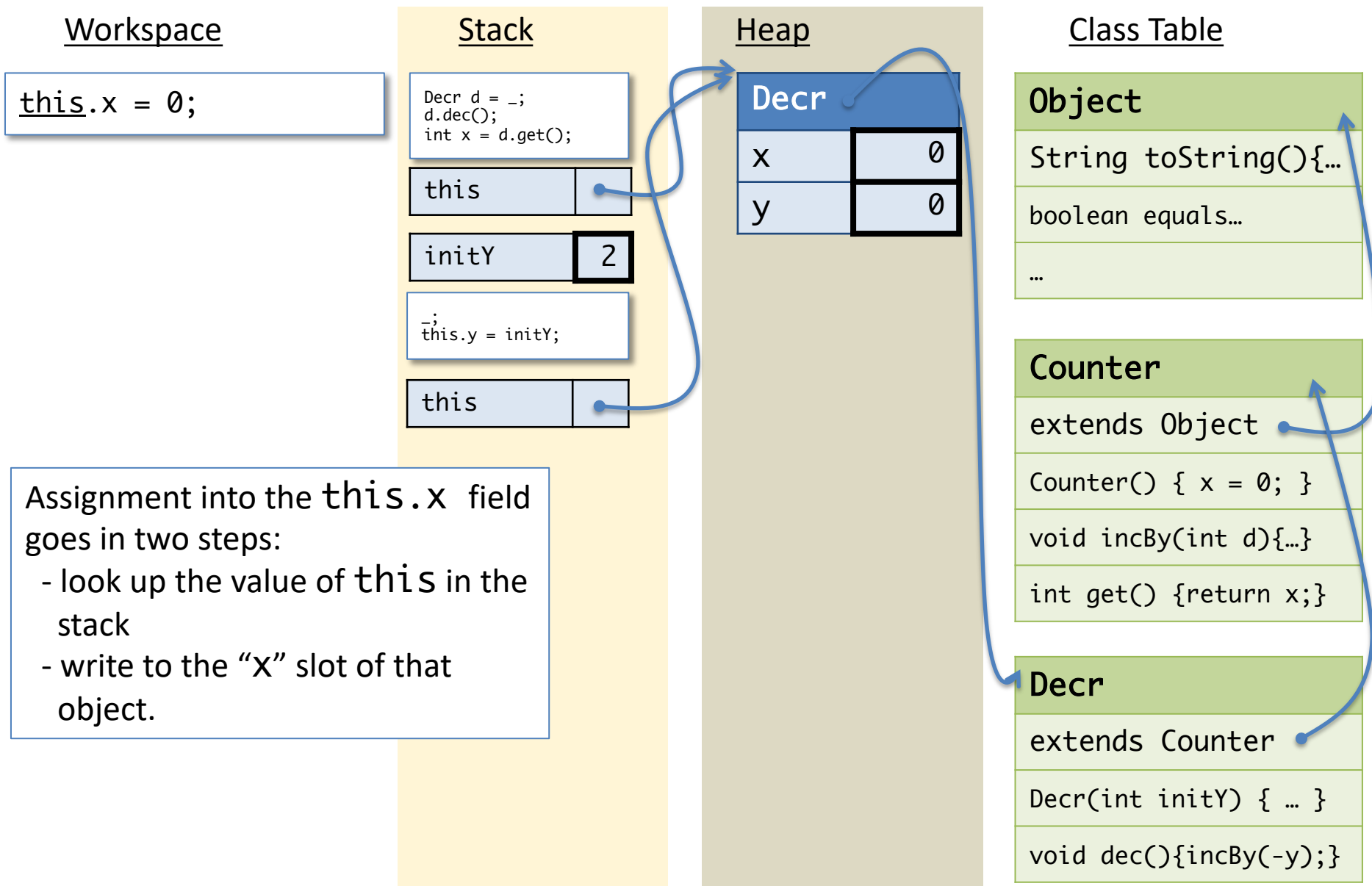
- The constructor (implicitly) calls the super constructor
- Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new this pointer.

Abstract Stack Machine



(Running Object's default constructor omitted.)

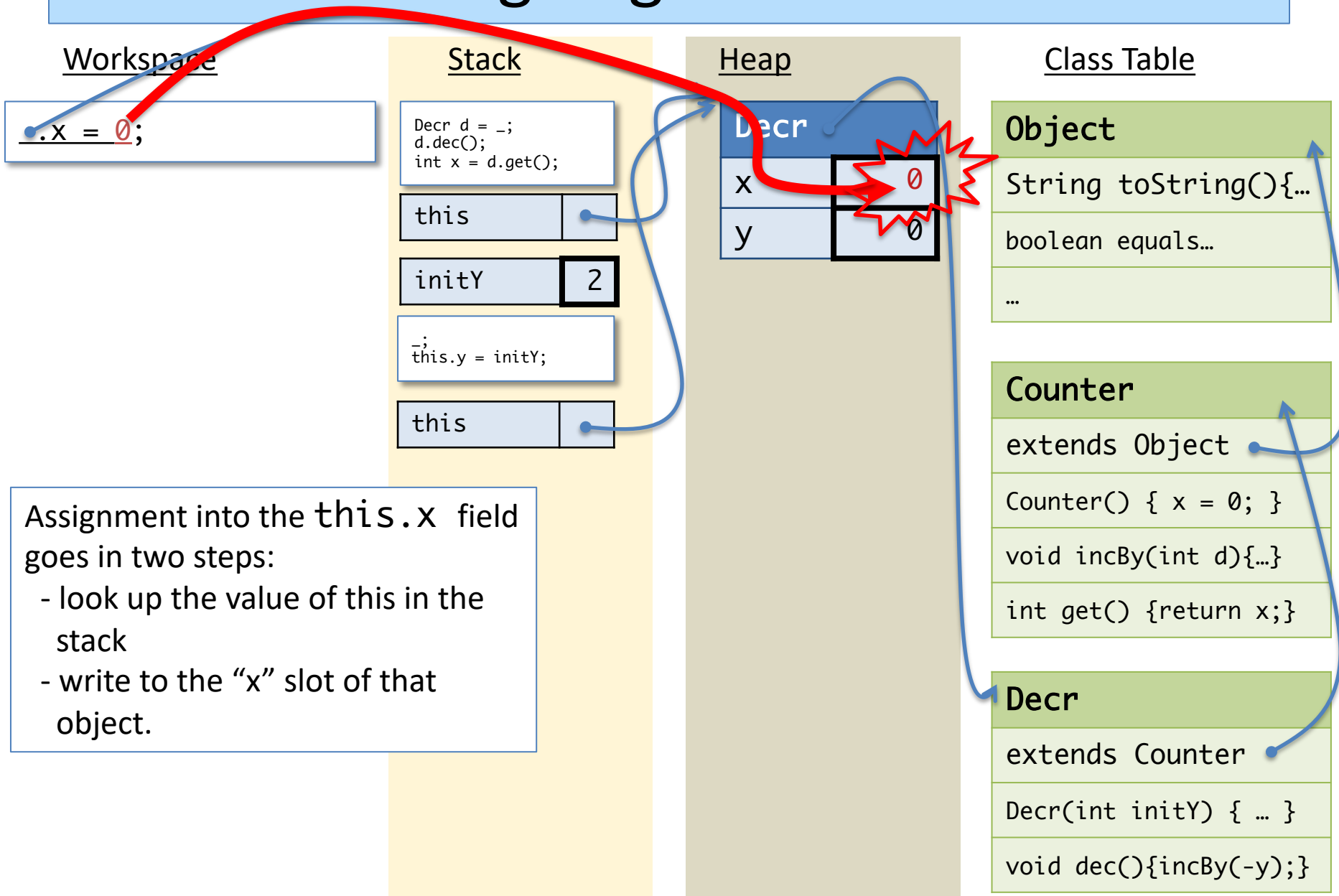
Assigning to a Field



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.

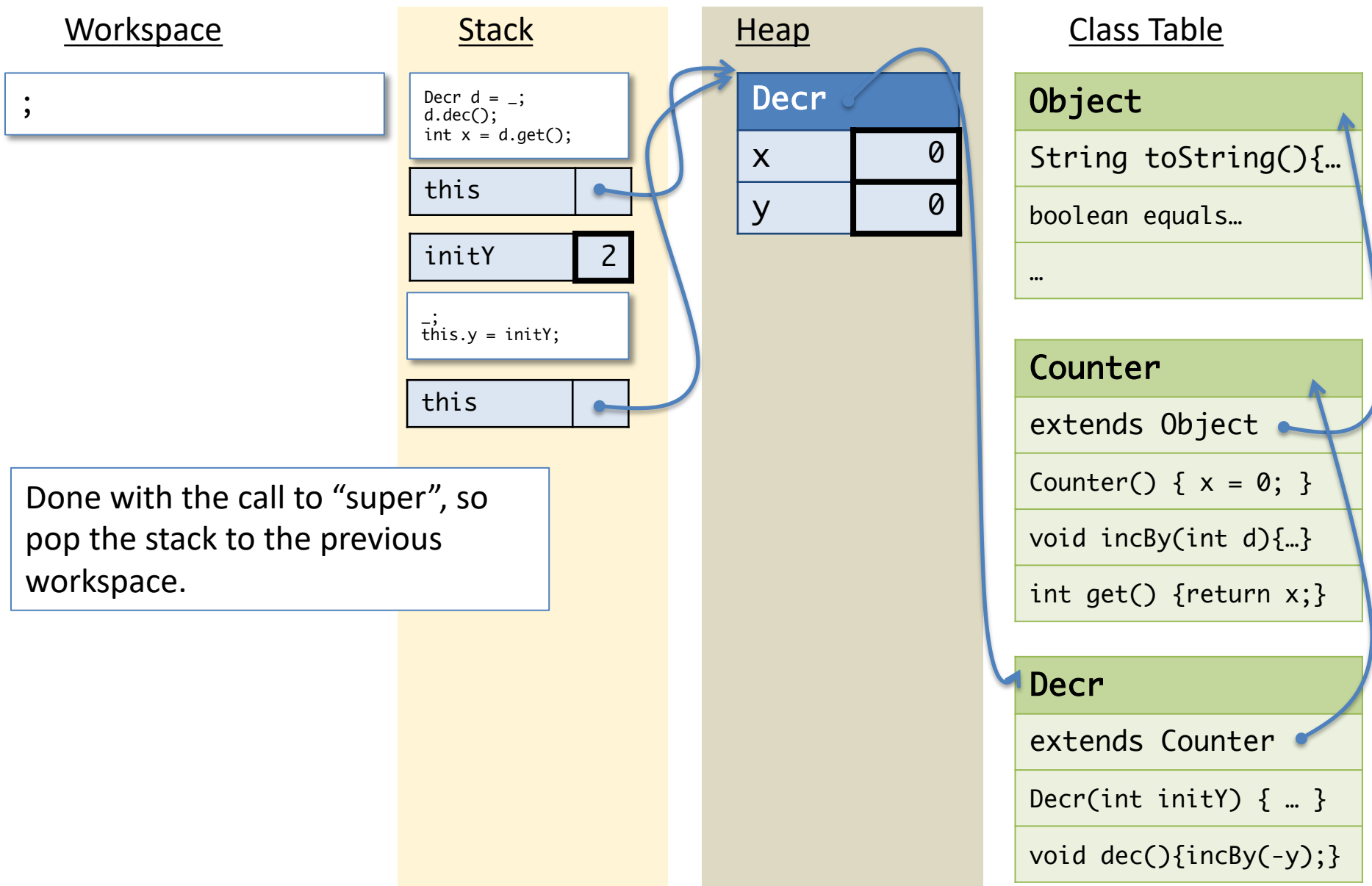
Assigning to a Field



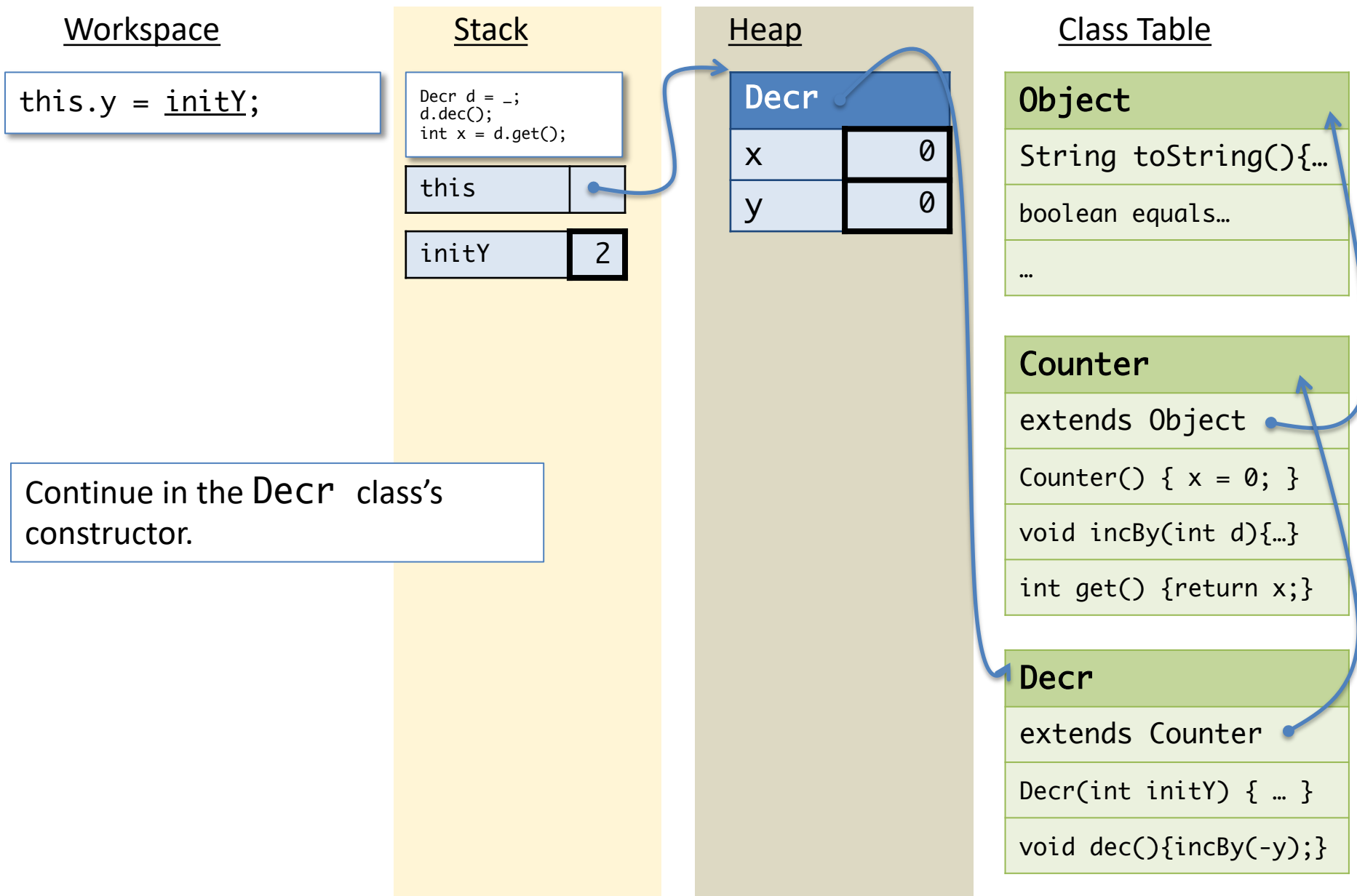
Assignment into the `this.x` field goes in two steps:

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

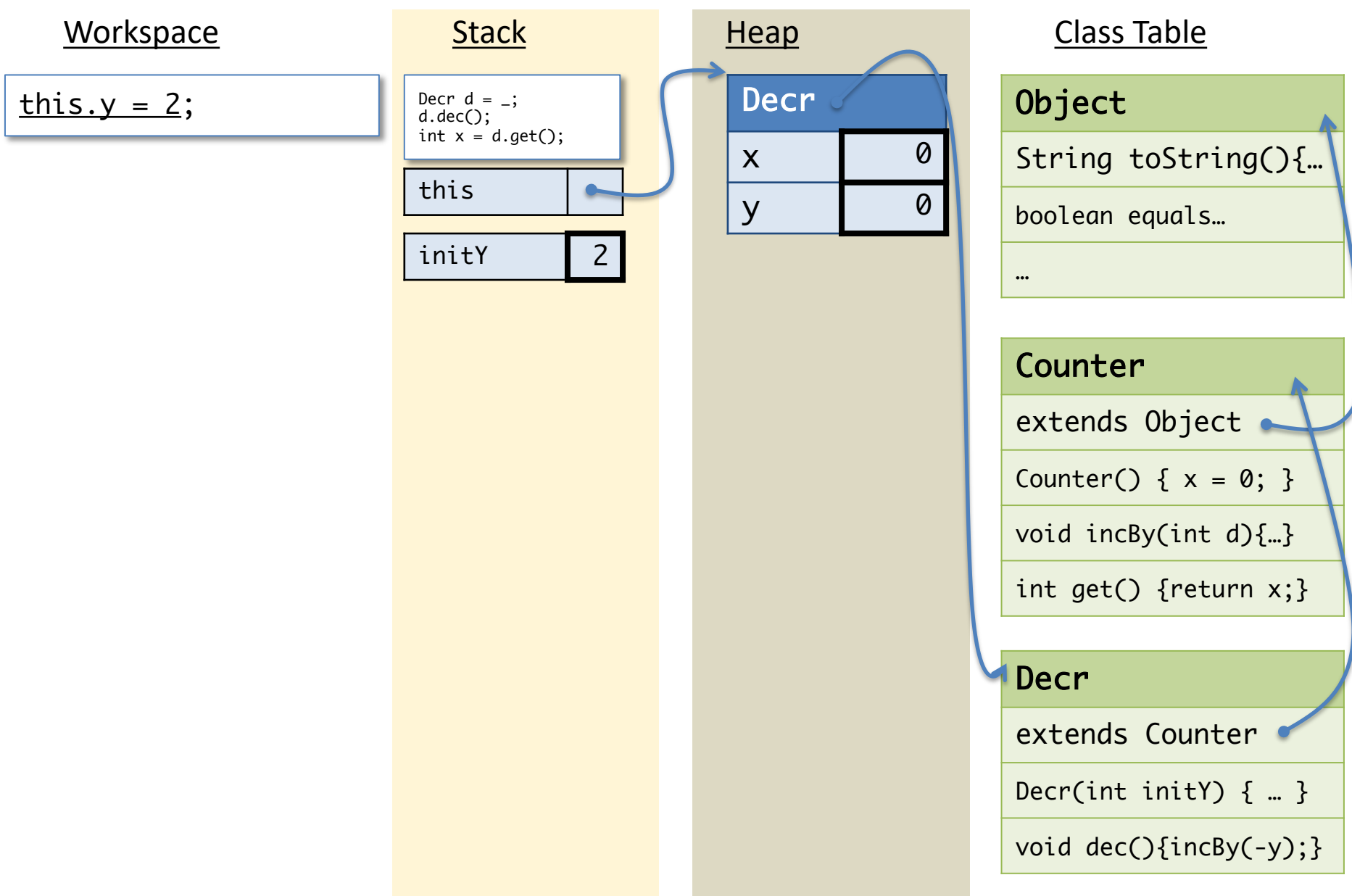
Done with the call



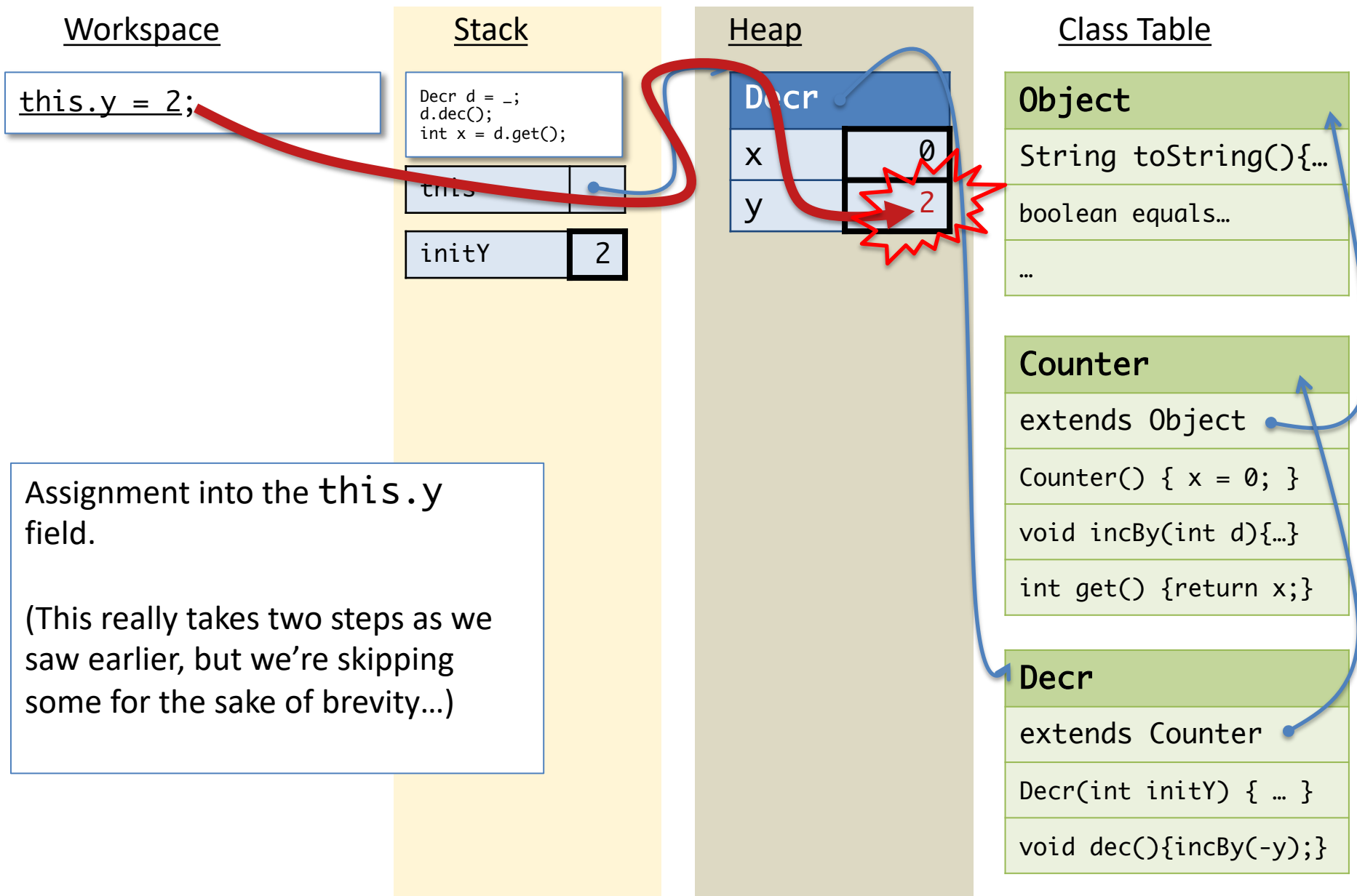
Continuing


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



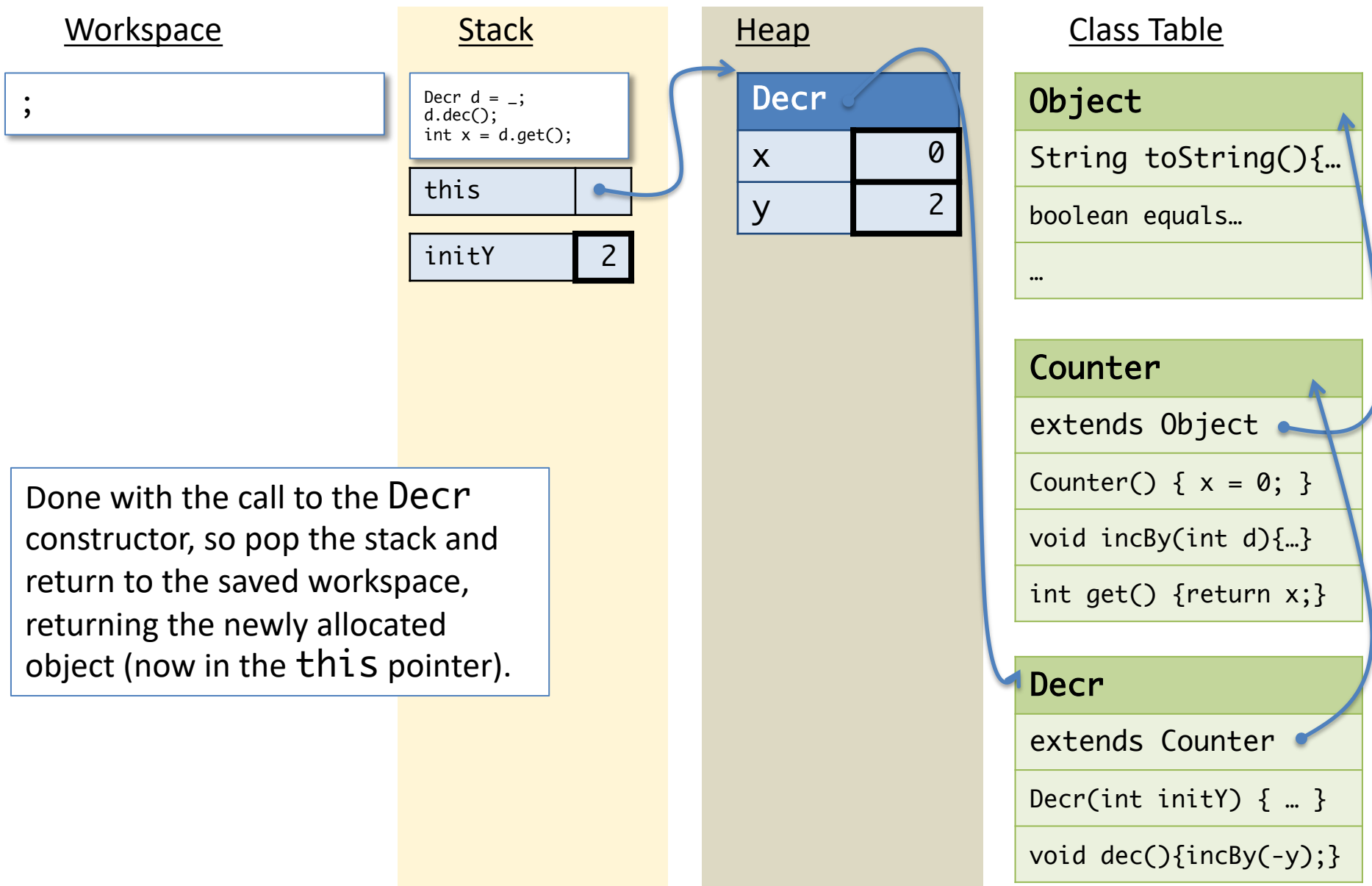
Assigning to a field



Assignment into the `this.y` field.

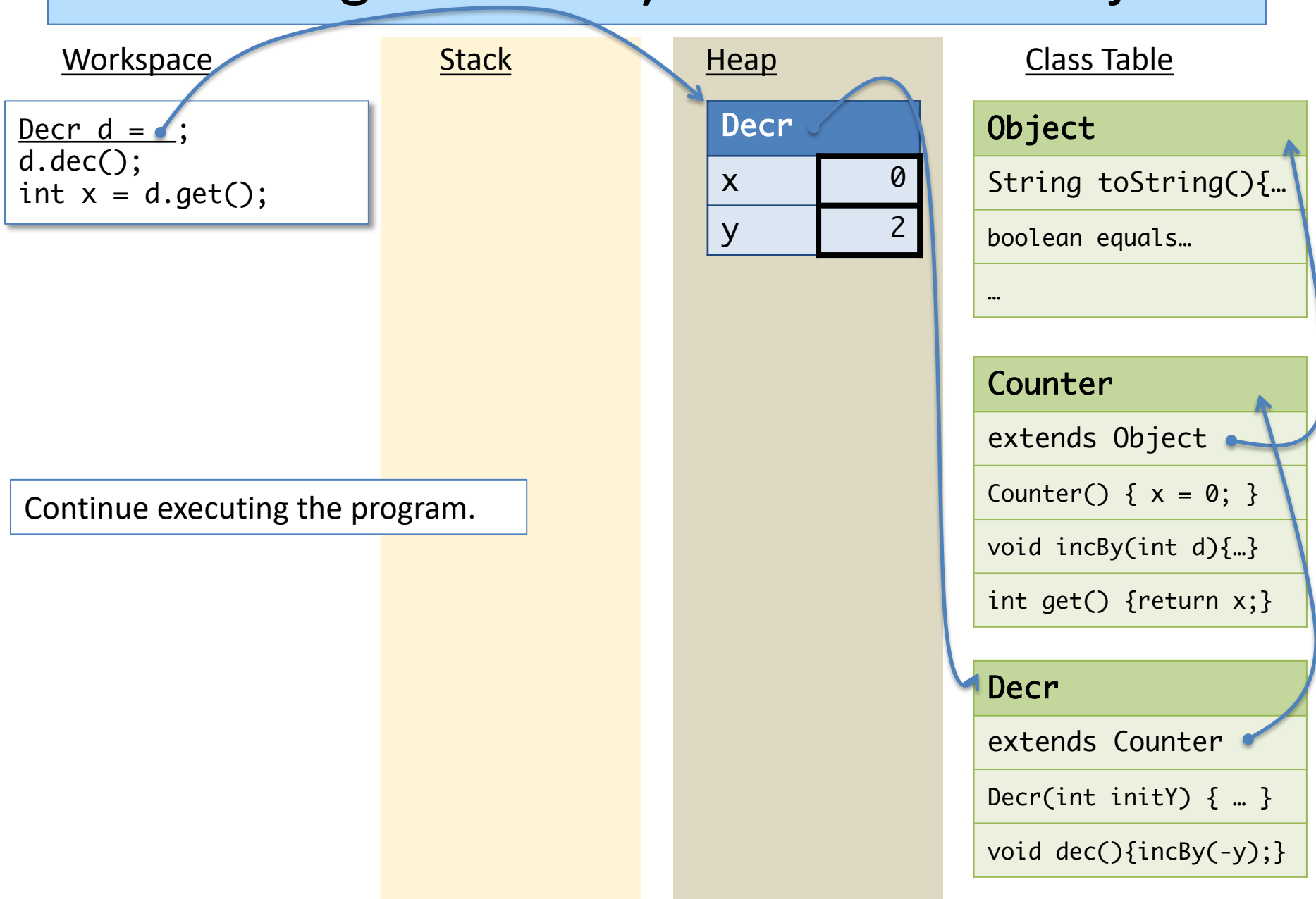
(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity…)

Done with the call

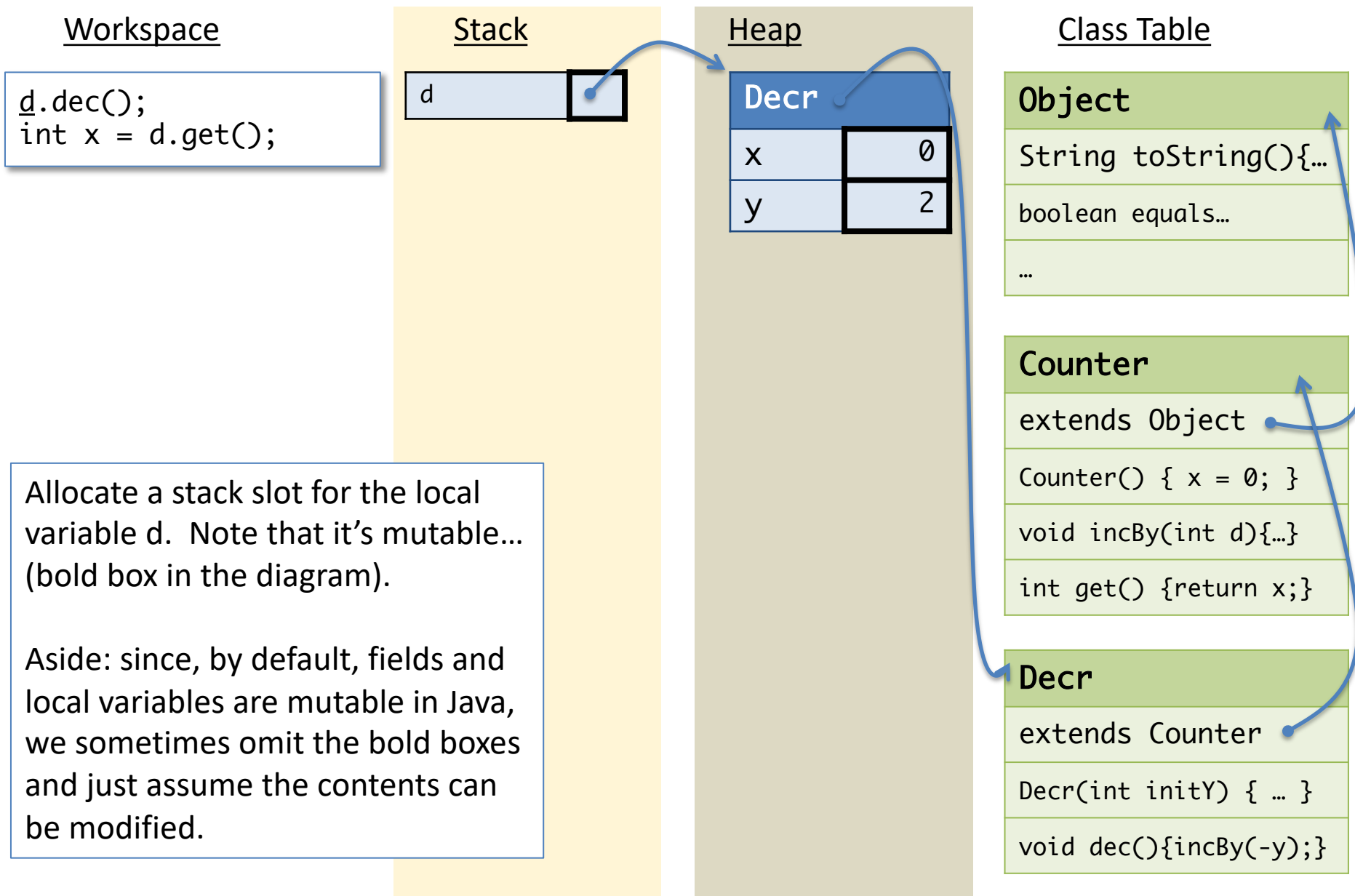


Done with the call to the `Decr` constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the `this` pointer).

Returning the Newly Constructed Object



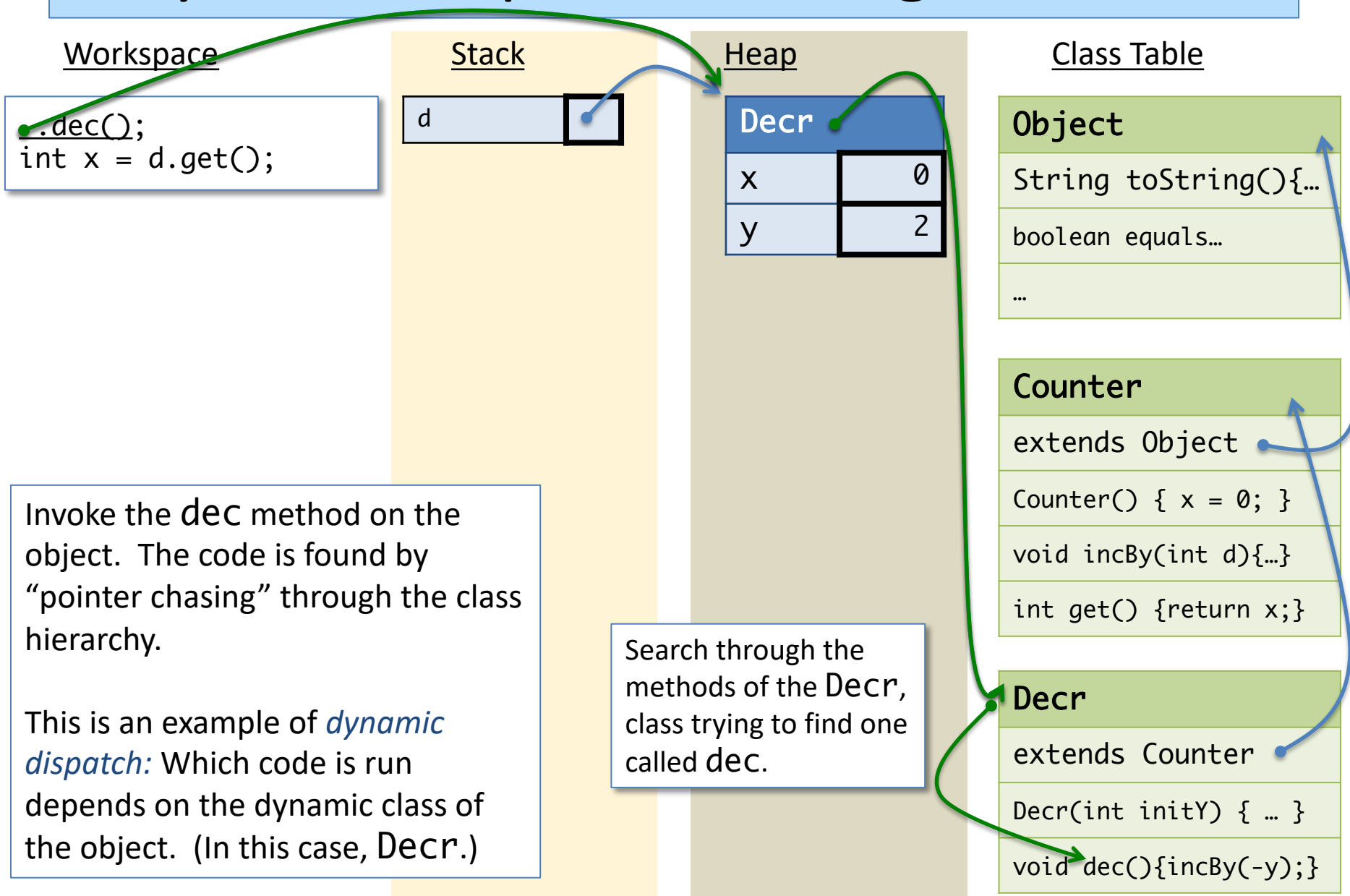
Allocating a local variable



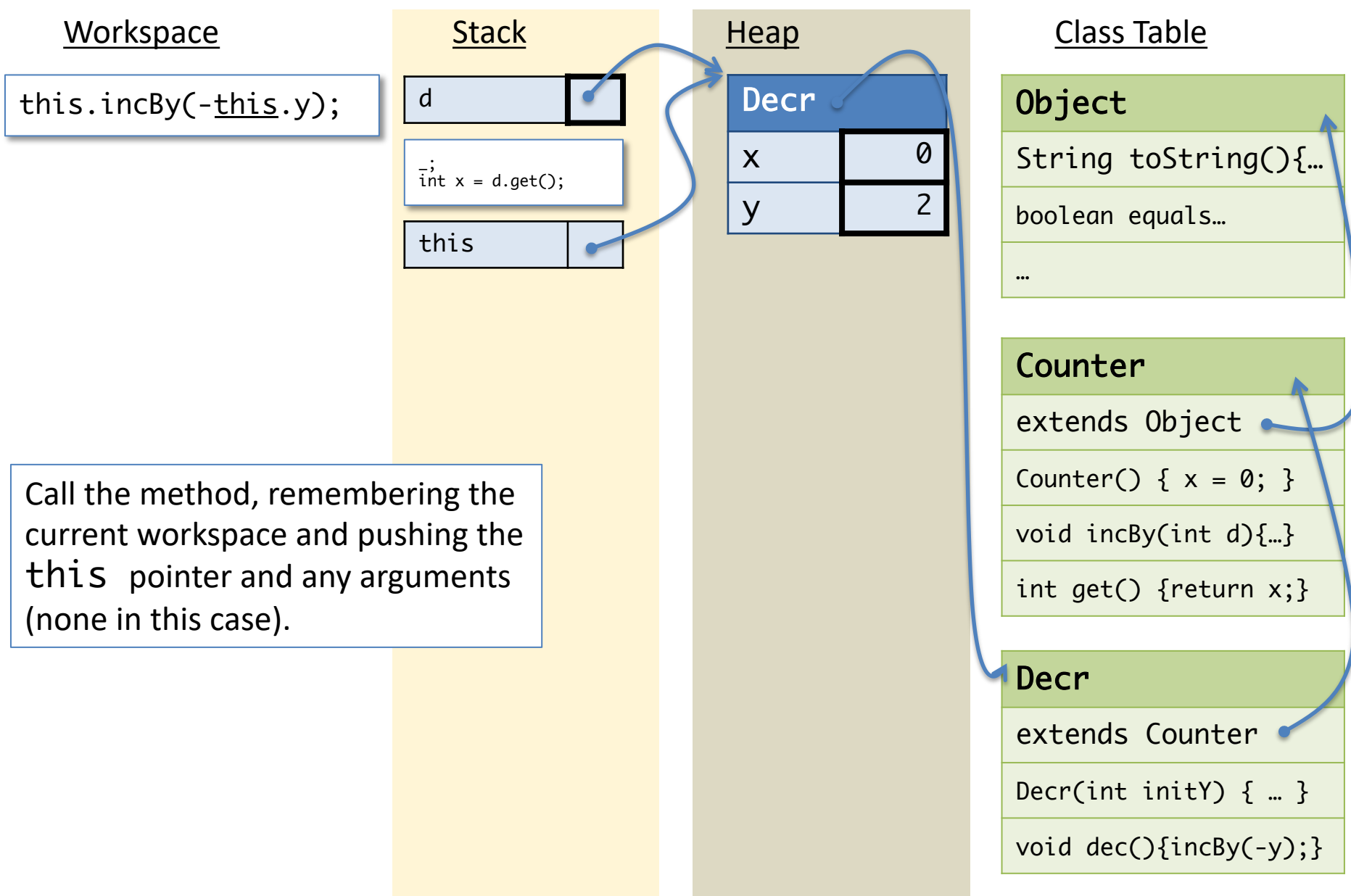
Allocate a stack slot for the local variable d. Note that it's mutable… (bold box in the diagram).

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

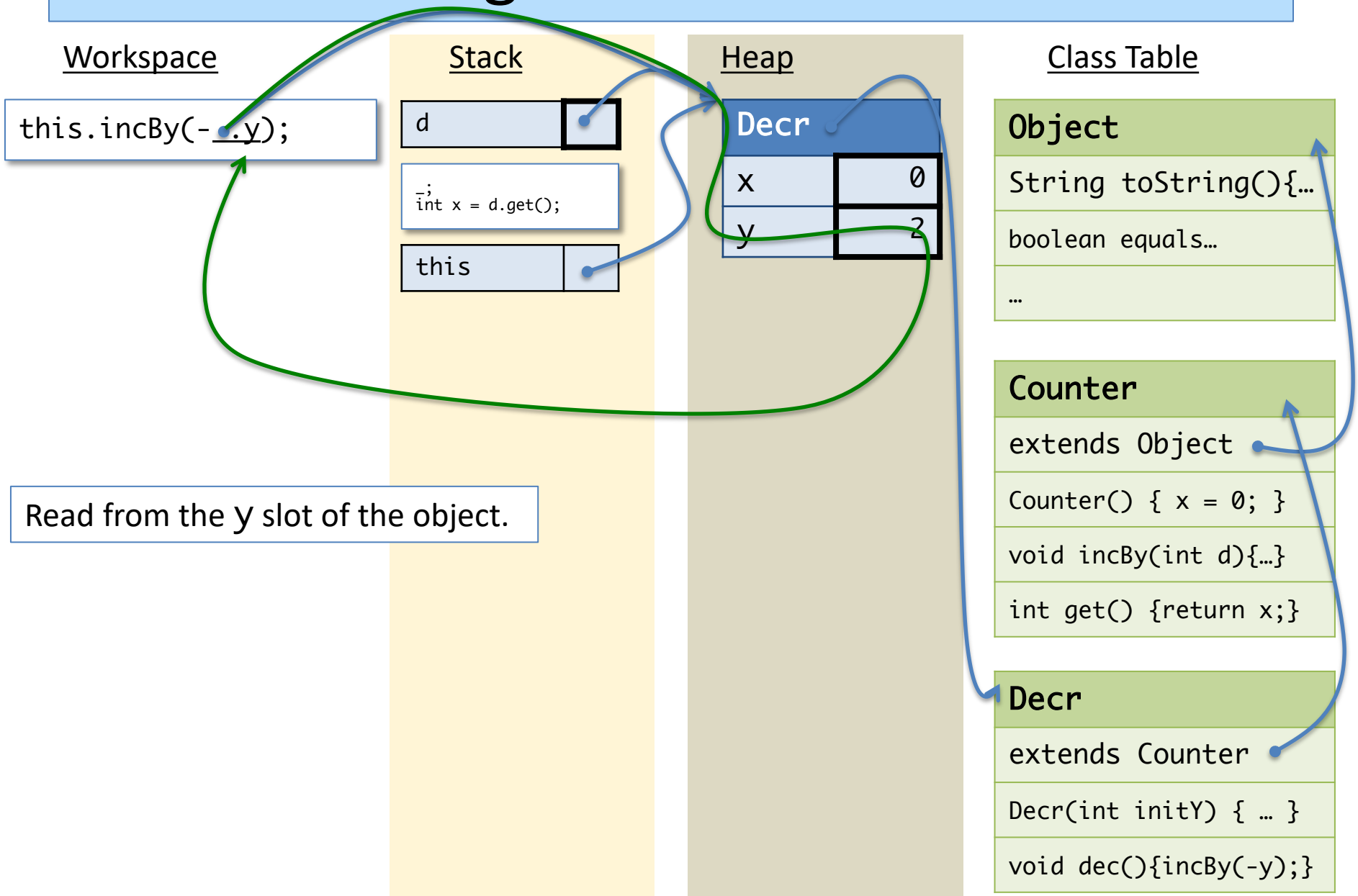


Dynamic Dispatch: Finding the Code



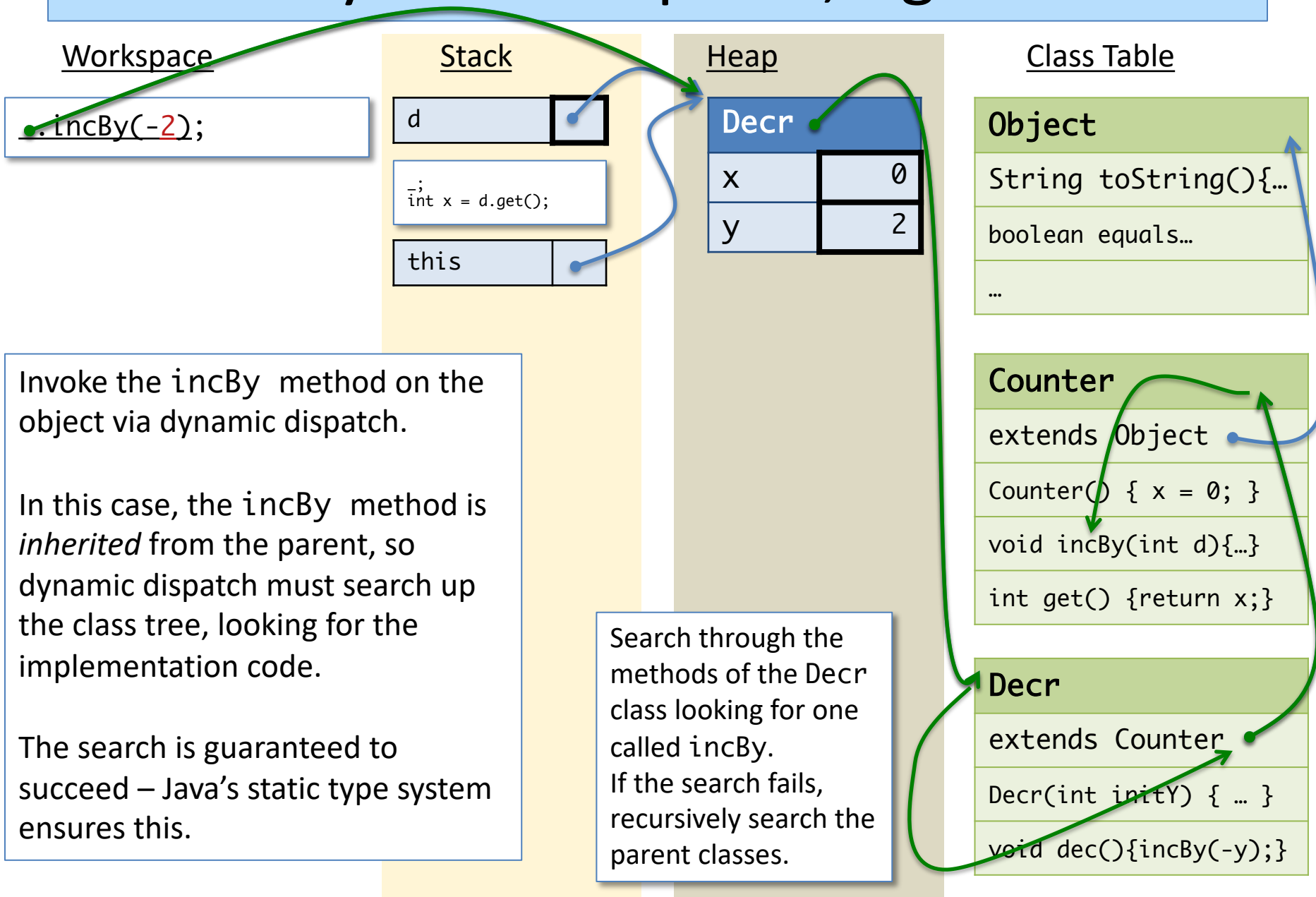
Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

Reading a Field's Contents

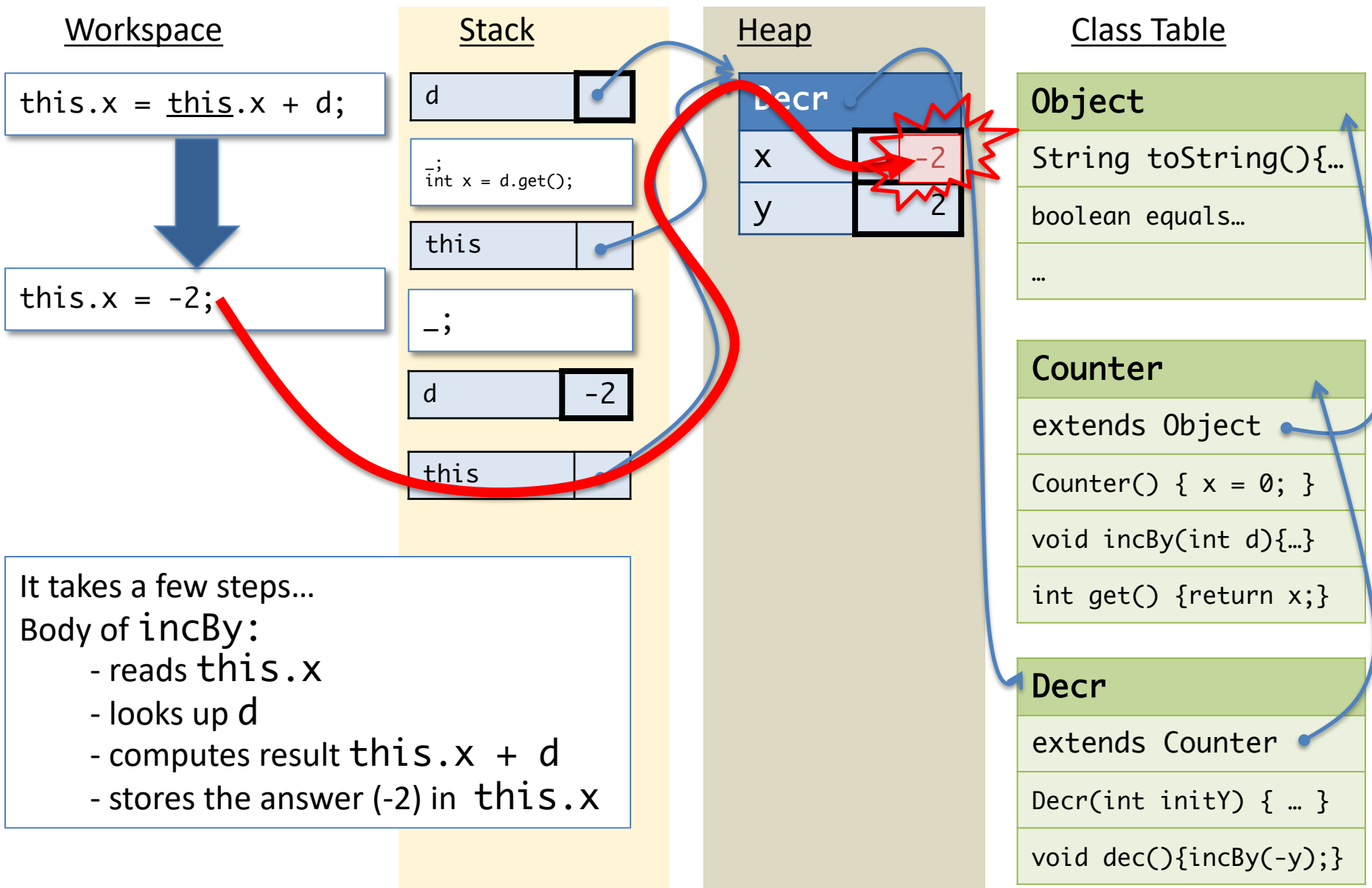


Read from the y slot of the object.

Dynamic Dispatch, Again



Running the body of incBy



It takes a few steps…

Body of `incBy`:

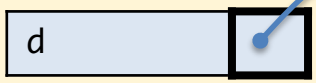
- reads `this.x`
- looks up `d`
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

After a few more steps…

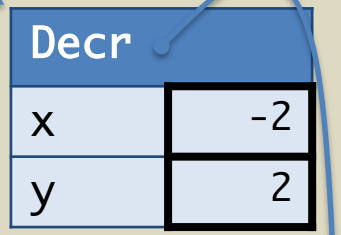
Workspace

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

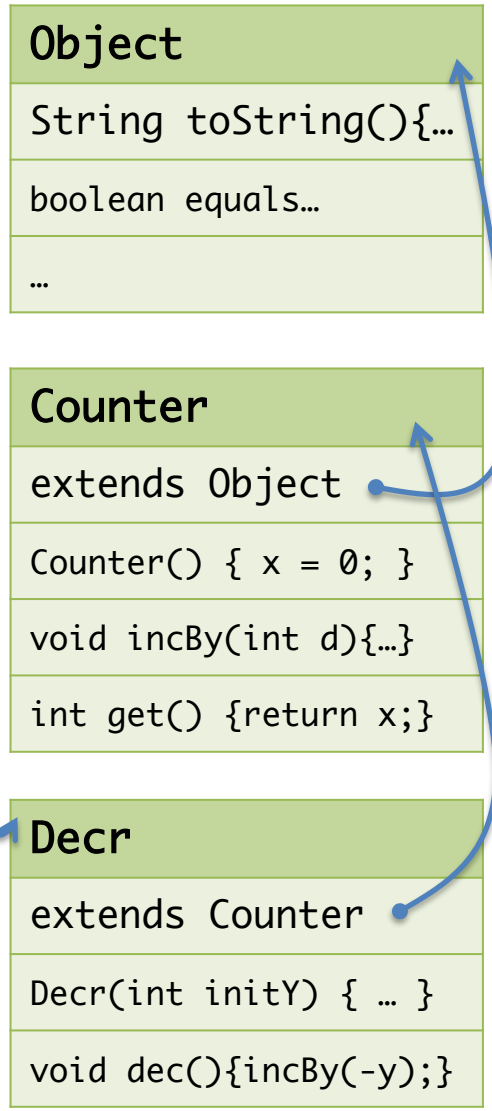
Stack



Heap

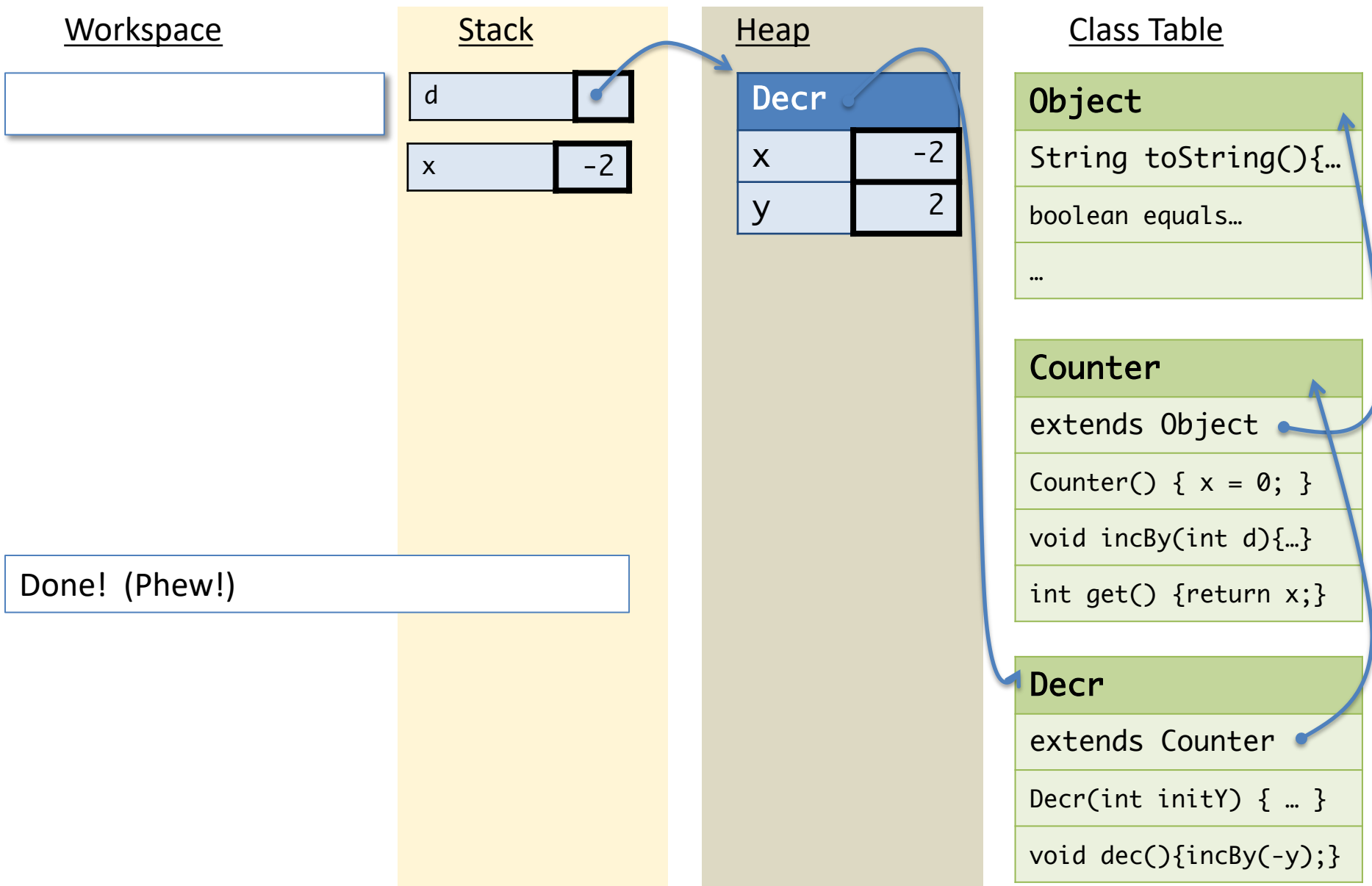


Class Table



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.

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 superclass 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 type (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.`

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