Programming Languages and Techniques (CIS120)

Lecture 26

March 20, 2013

Dynamic Dispatch and Generics

Announcements

- HW08 (Text Adventure) is due Monday at 11:59:59pm
- *Midterm 2 is Friday, March 29th in class*
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in OCaml)
  - Arrays in (Java)
  - Subtyping & Inheritance (in Java)
- Practice exams available on course website

The Java Abstract Stack Machine and the Class Table

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

Example

```
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

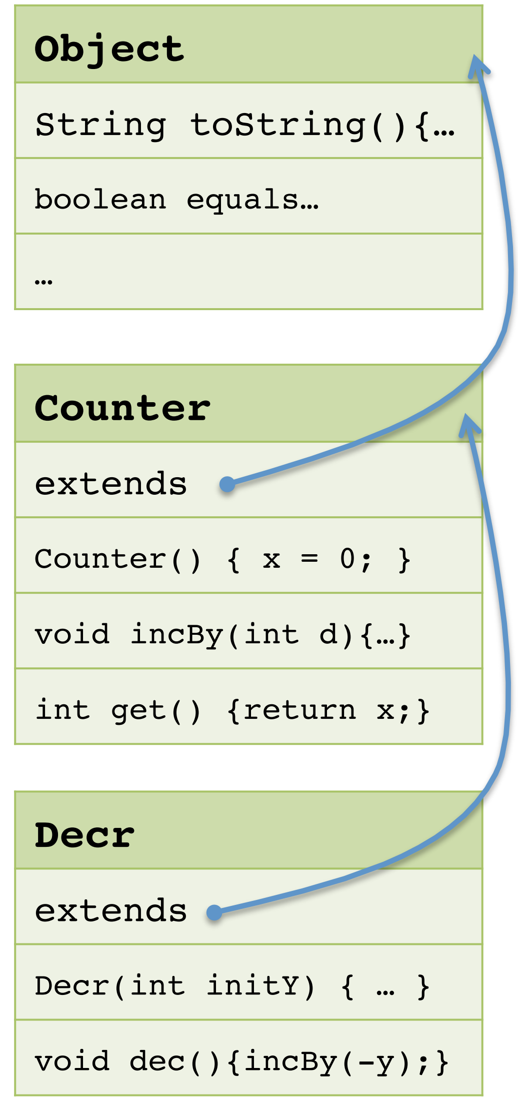
Workspace

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

Stack

Heap

Class Table



Allocating Space on the Heap

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;}

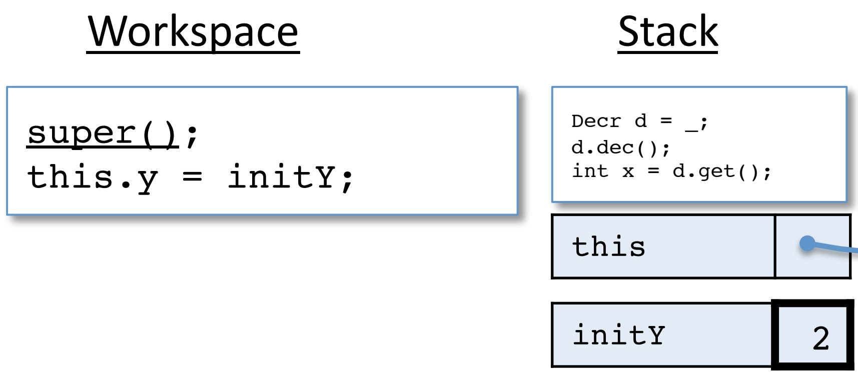
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}

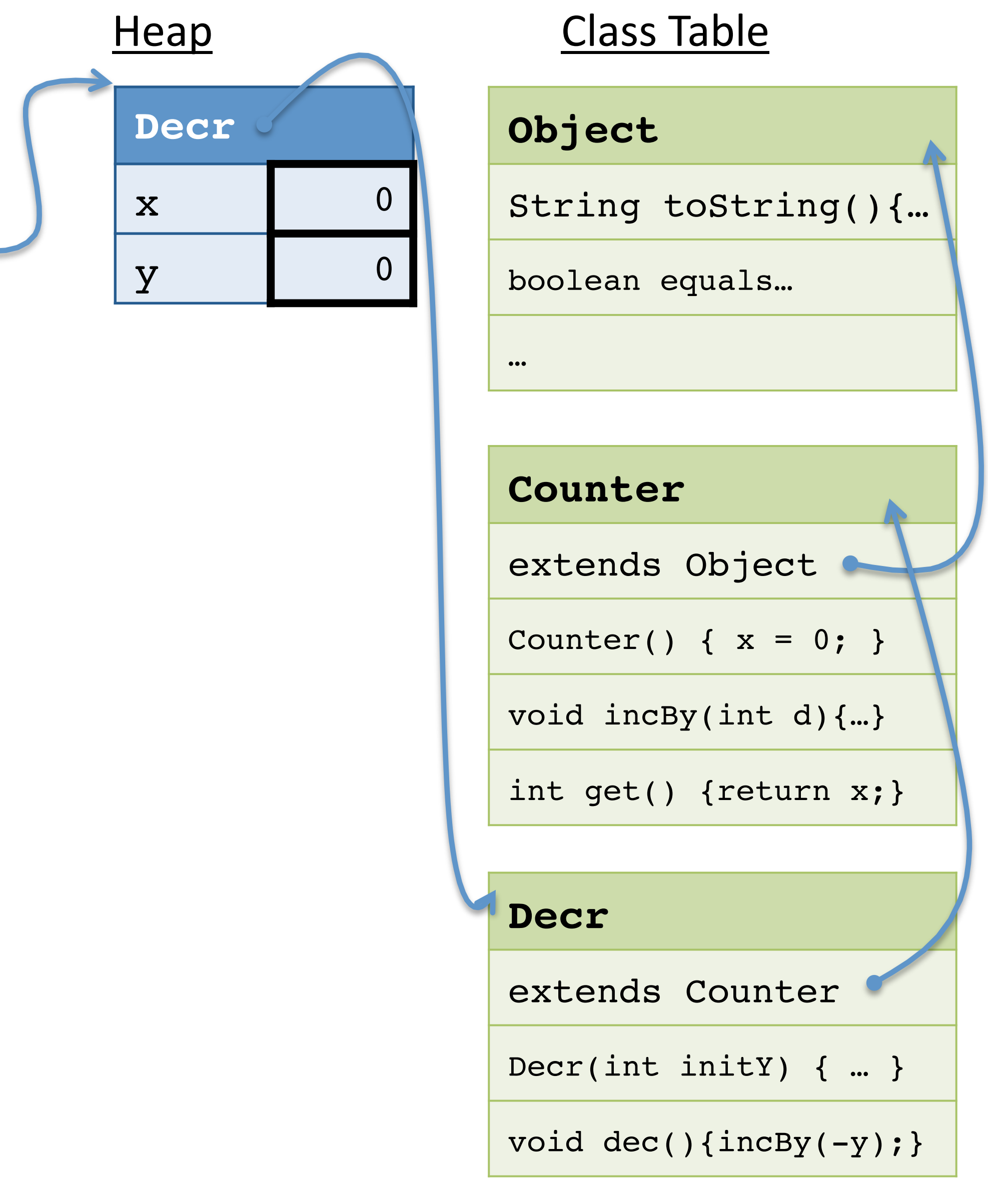
Invoking a constructor:

- 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

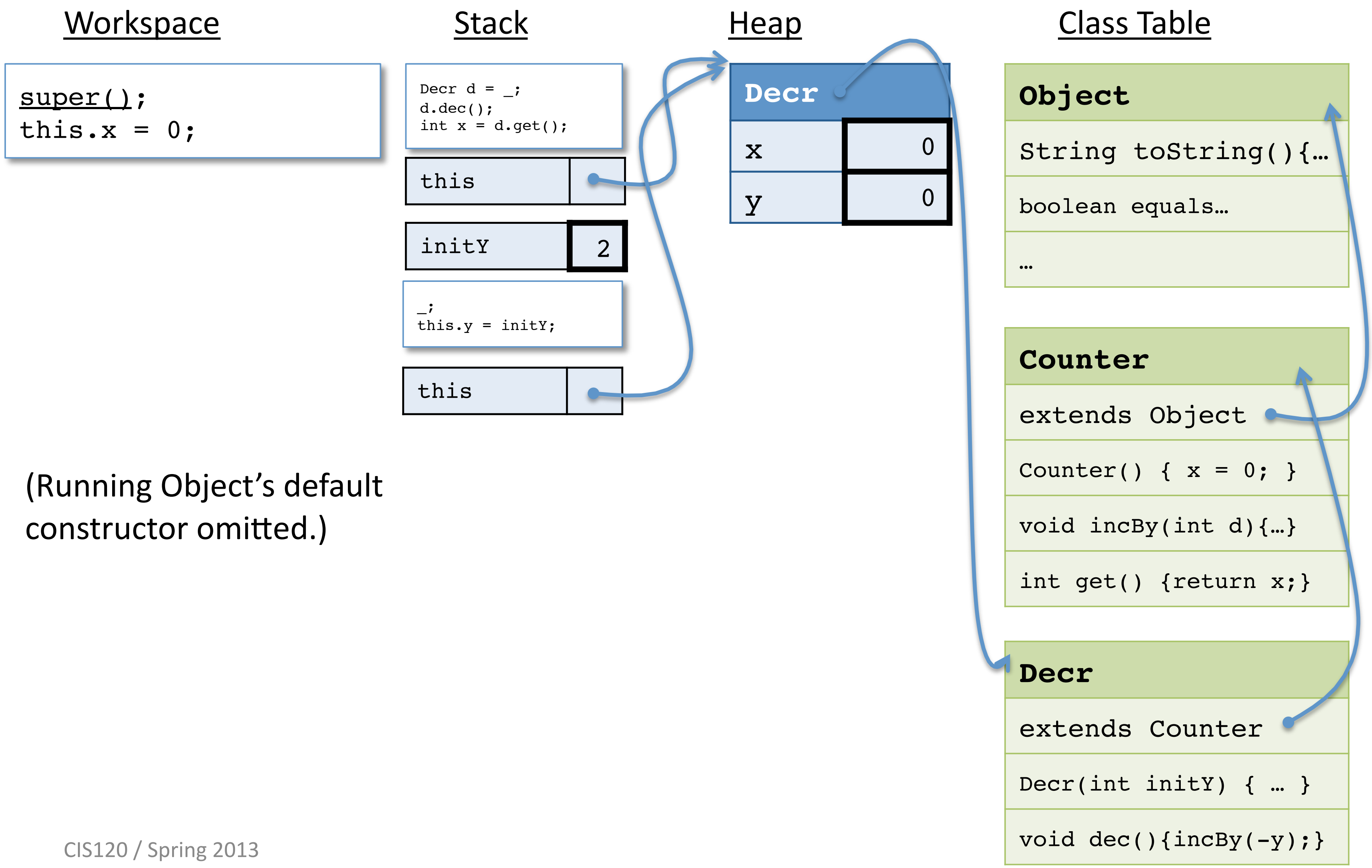




Call to `super`:

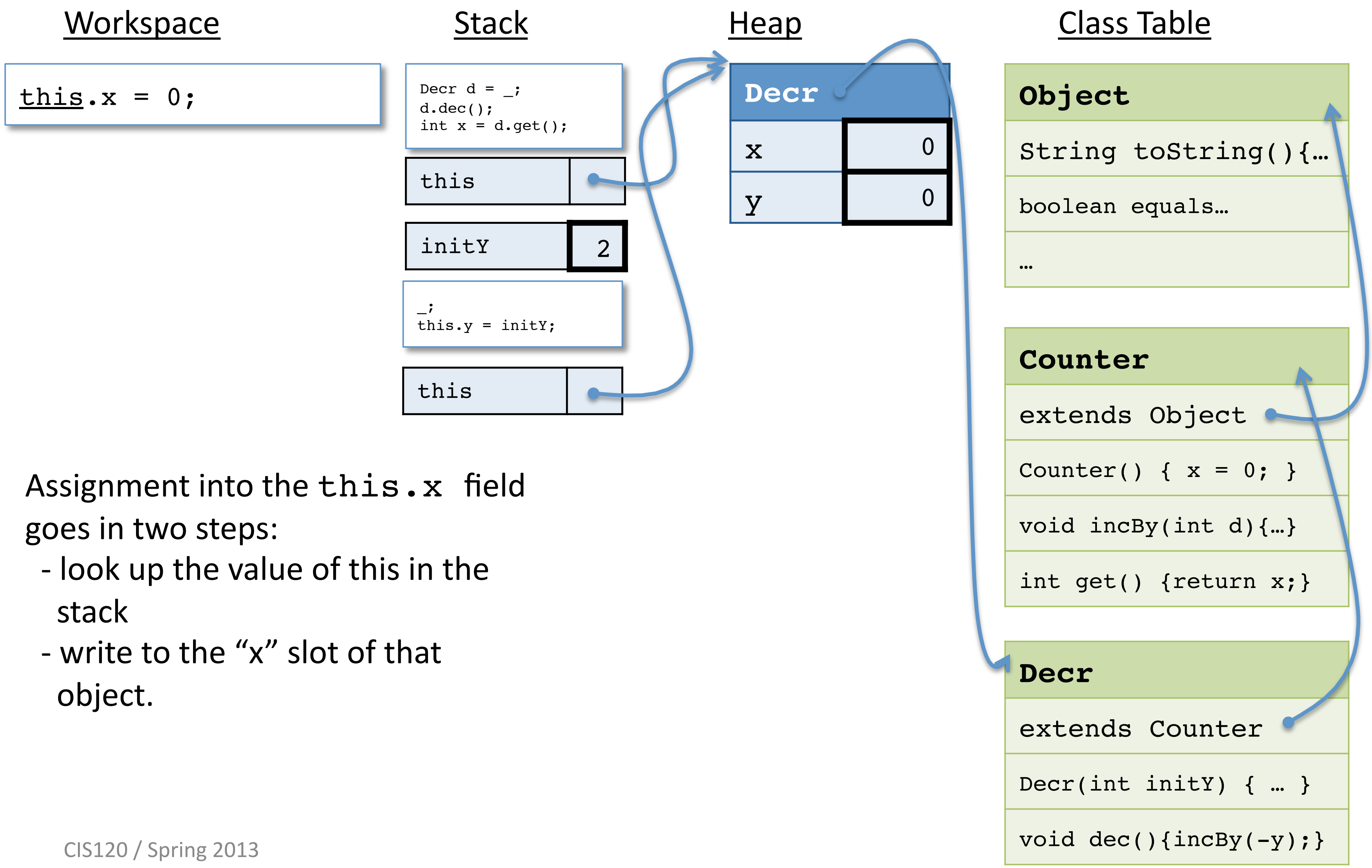
- The constructor (implicitly) calls the super constructor
- Invoking a method/constructor pushes the saved workspace, the method params (none here) and a new `this` pointer.

Abstract Stack Machine



(Running Object's default constructor omitted.)

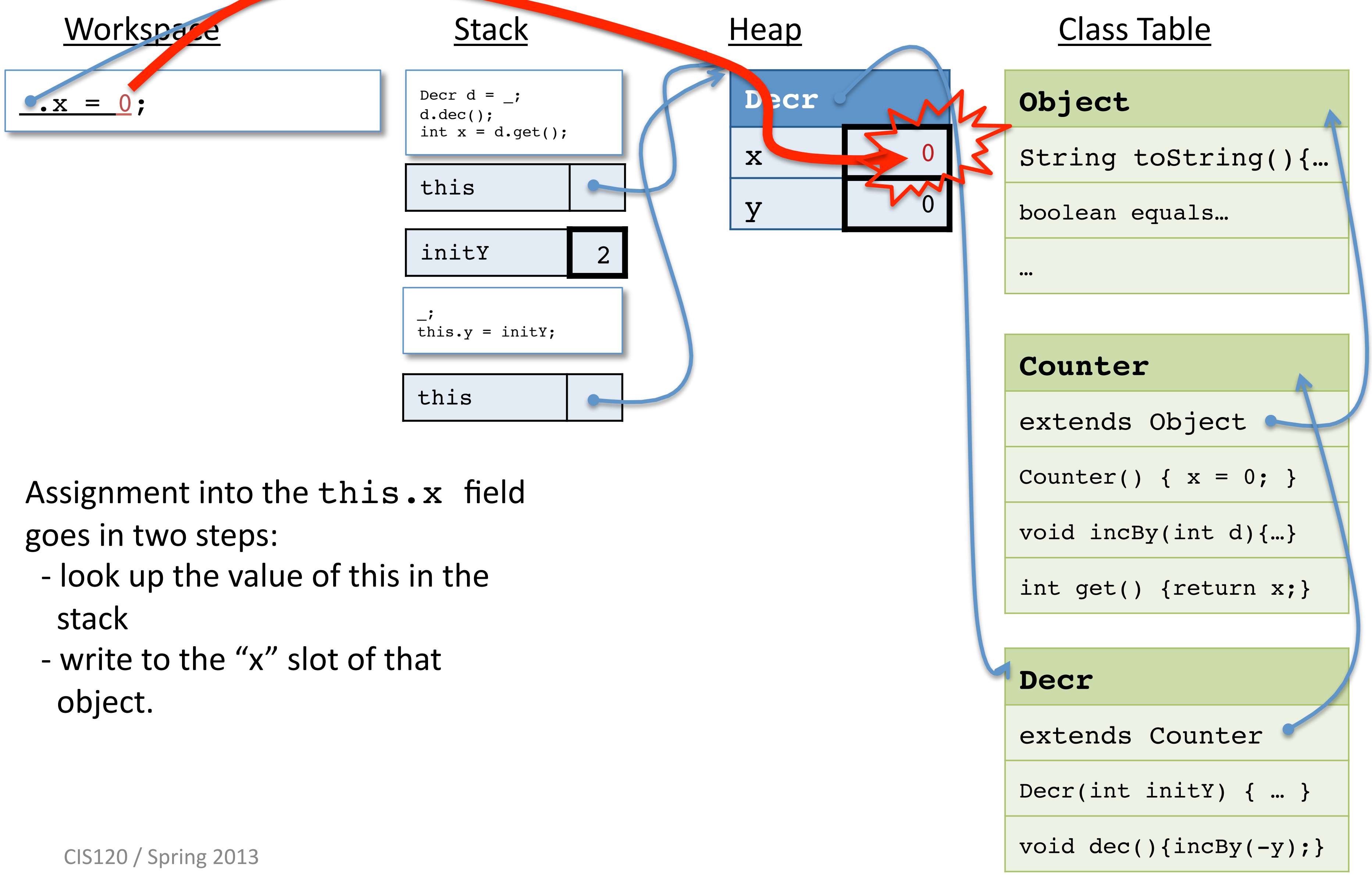
Assigning to a Field



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.

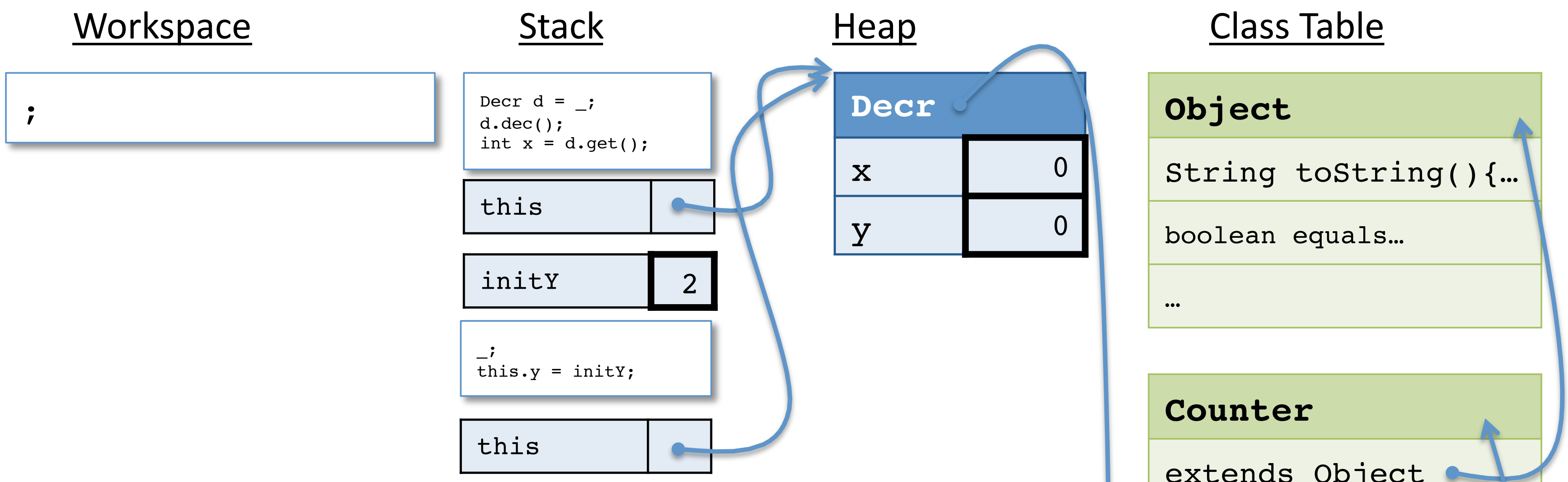
Assigning to a Field

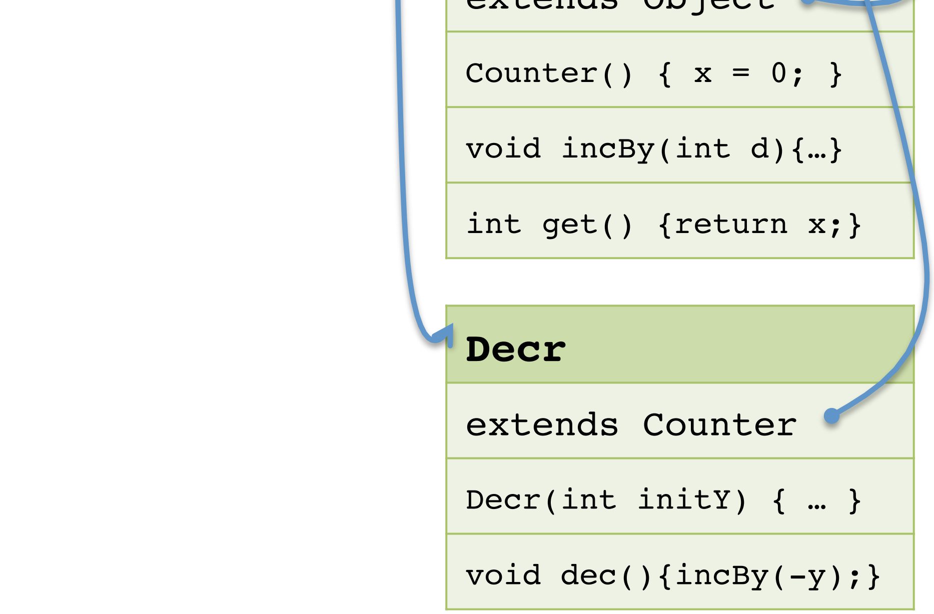


Assignment into the `this.x` field goes in two steps:

- 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

Workspace

```
this.y = initY;
```

Stack

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

this	•
initY	2

Heap

Decr •

x	0
y	0

Class Table

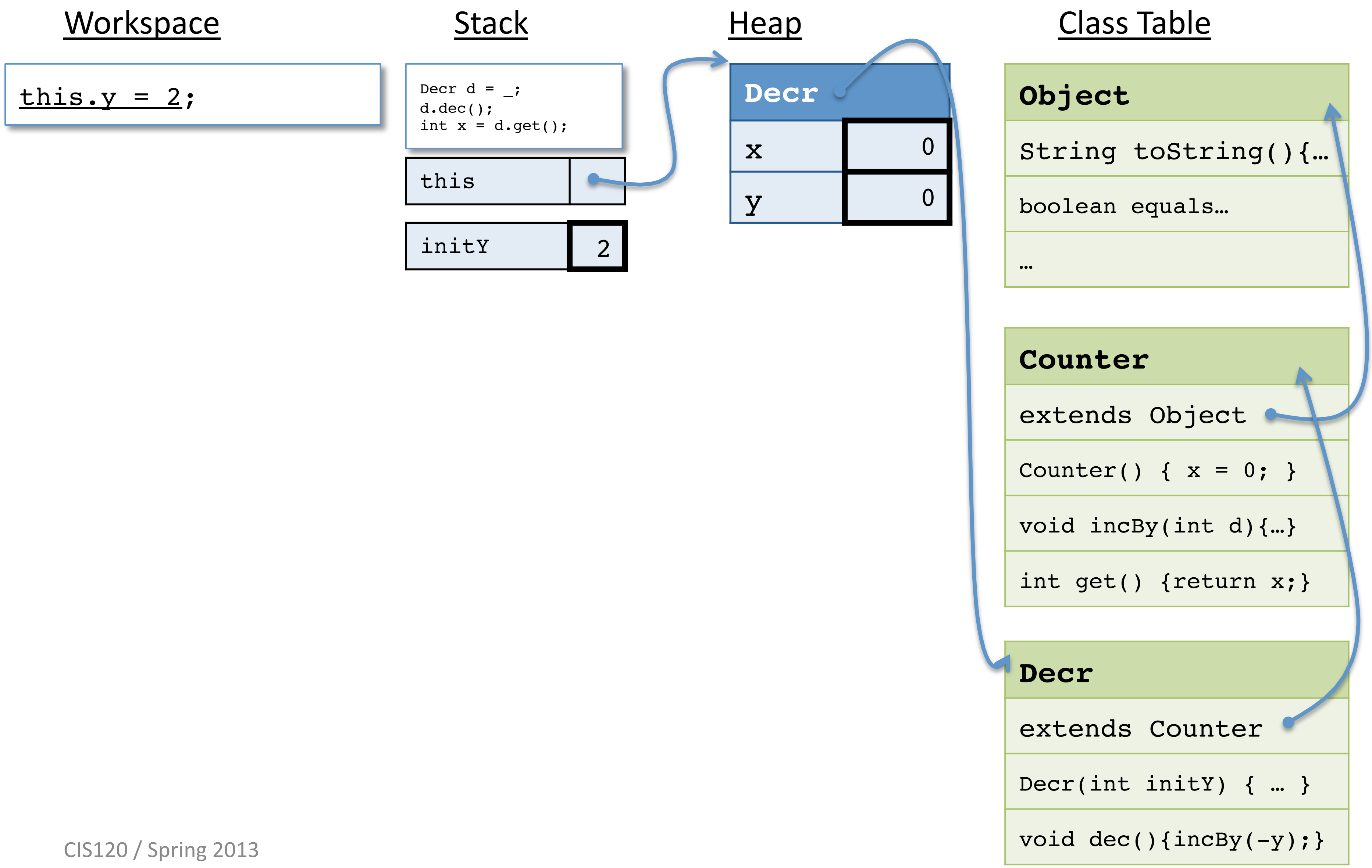
Object
String toString(){…
boolean equals…
…

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

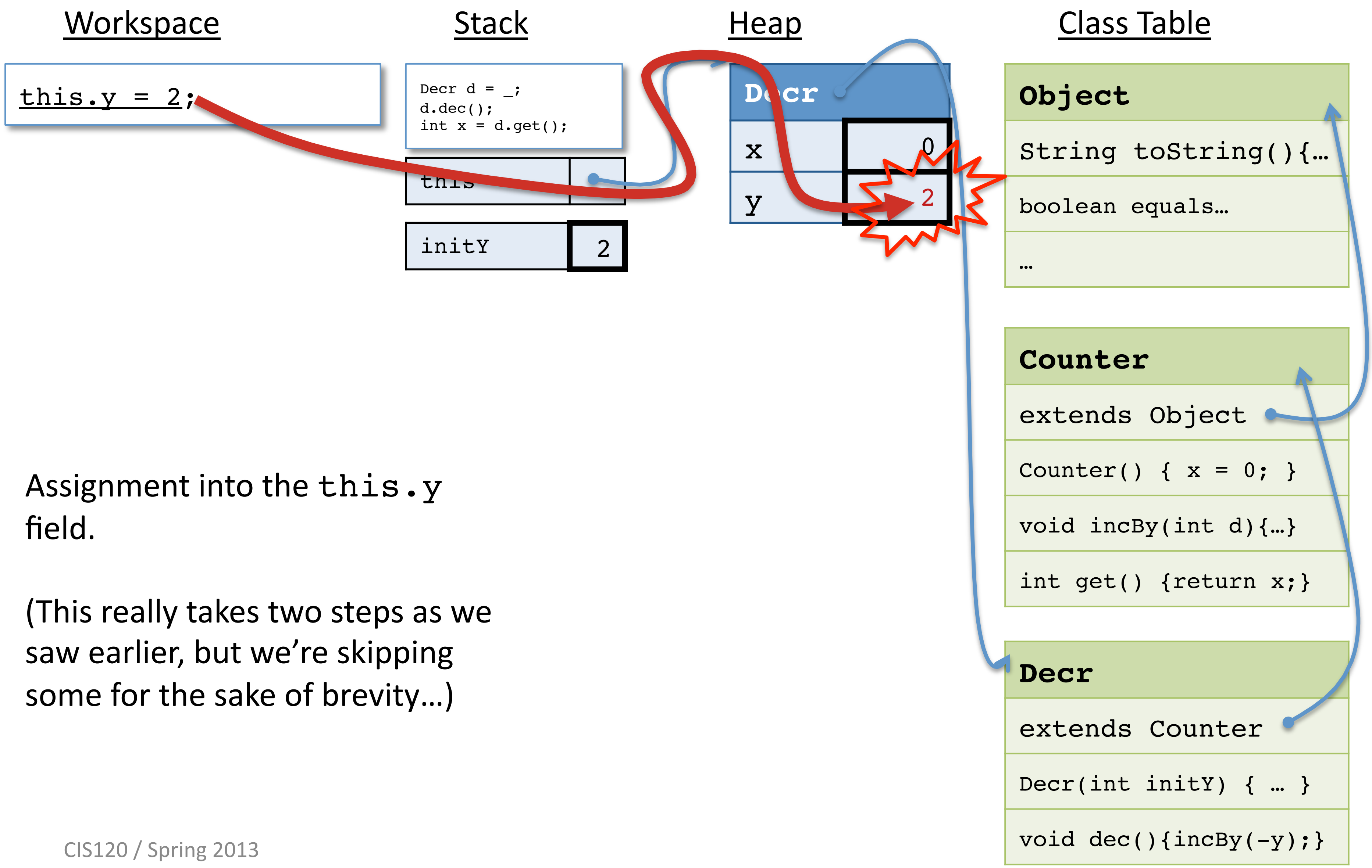
Decr
extends Counter •
Decr(int initY) { … }
void dec(){incBy(-y);}

Continue in the `Decr` class's constructor.

Abstract Stack Machine



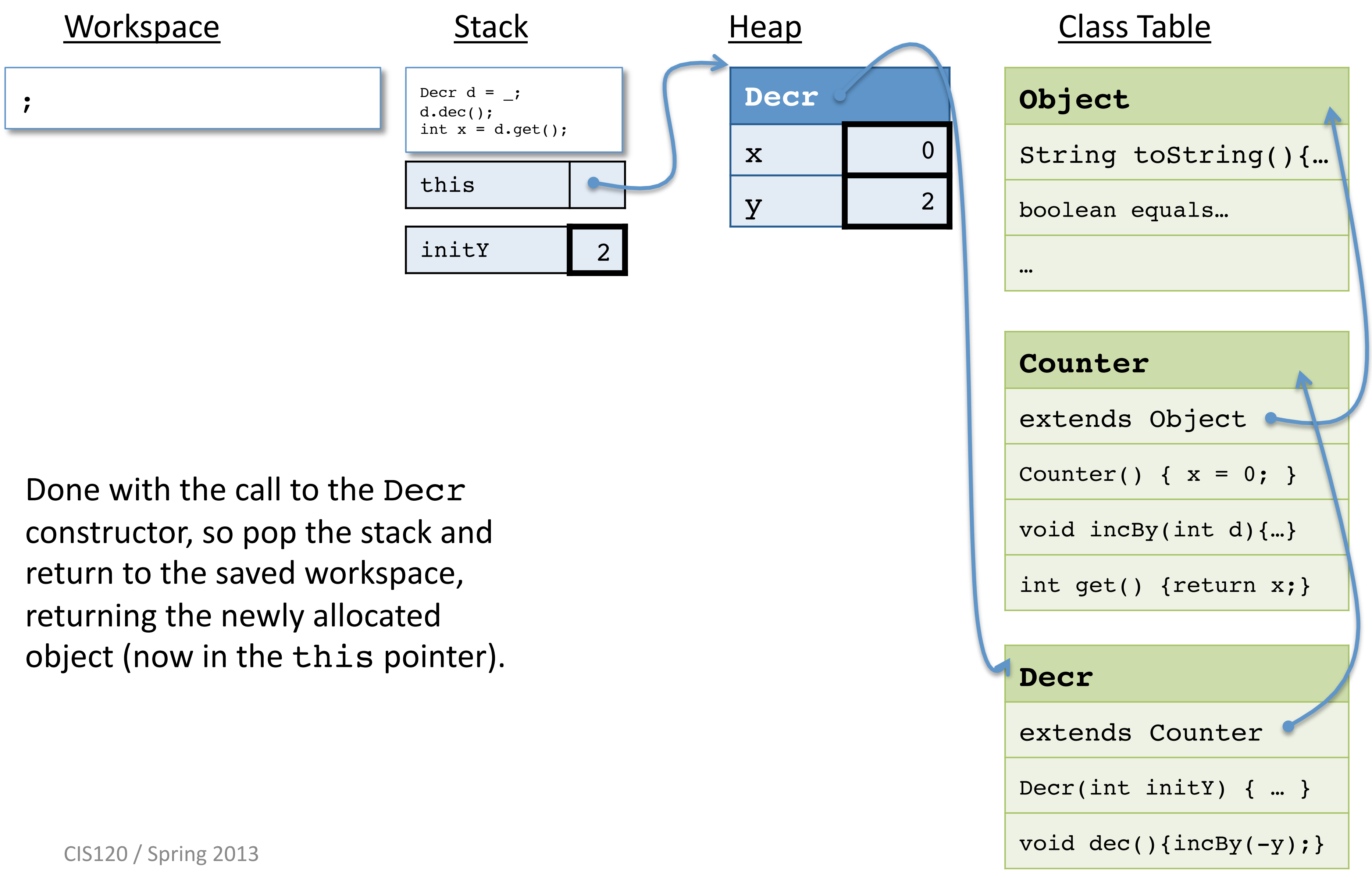
Assigning to a field



Assignment into the `this.y` field.

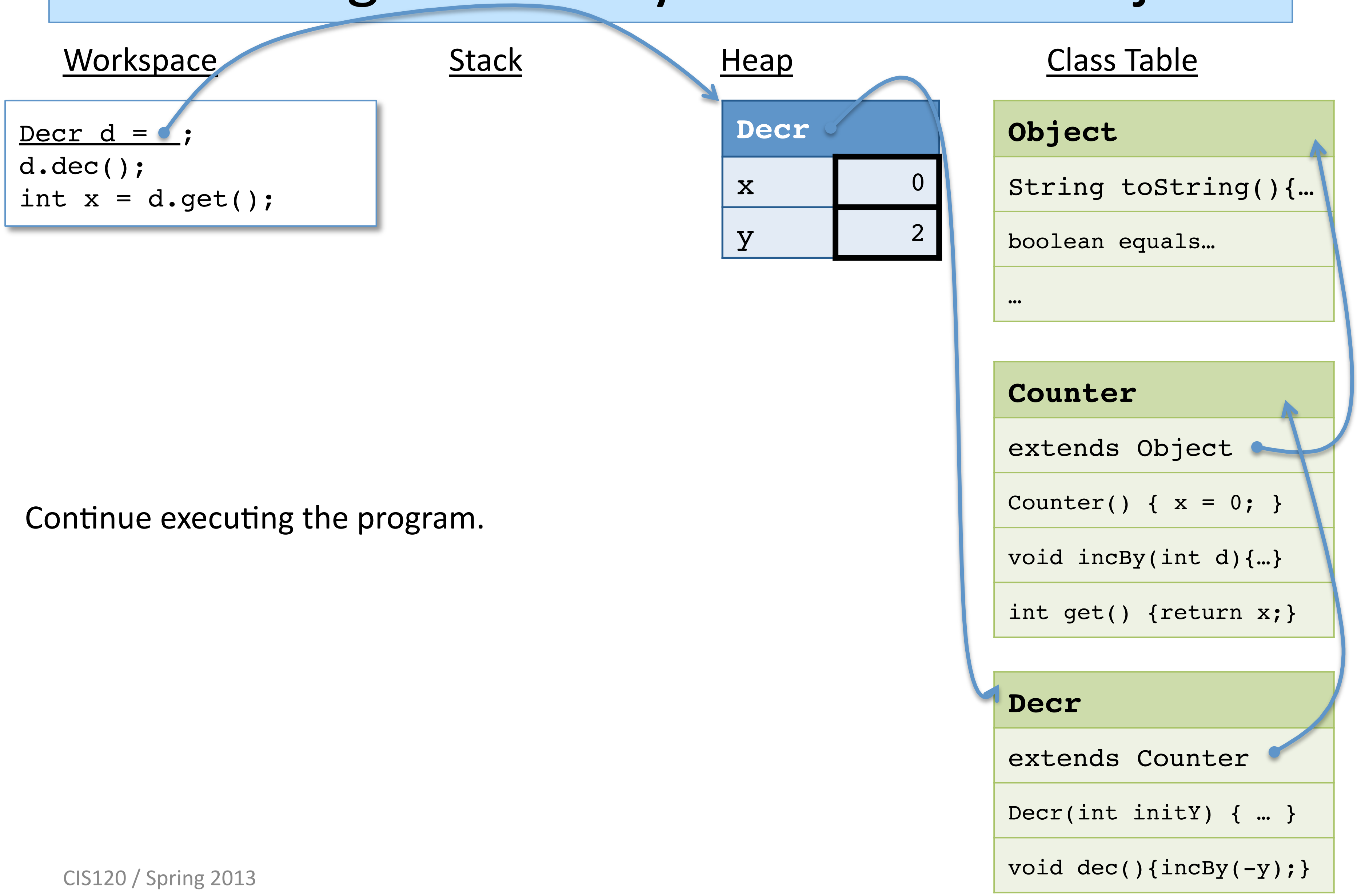
(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity…)

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

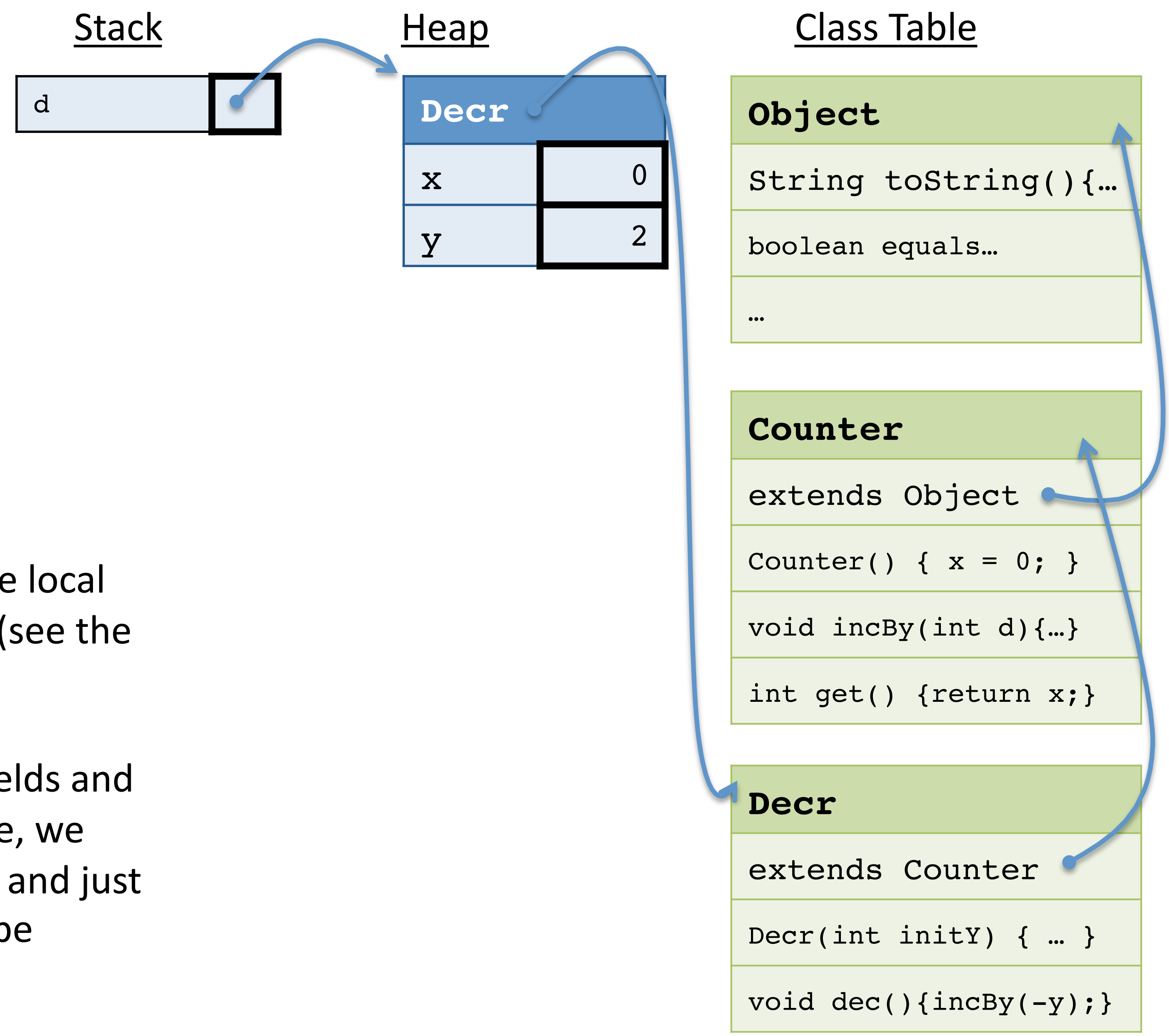


Continue executing the program.

Allocating a local variable

Workspace

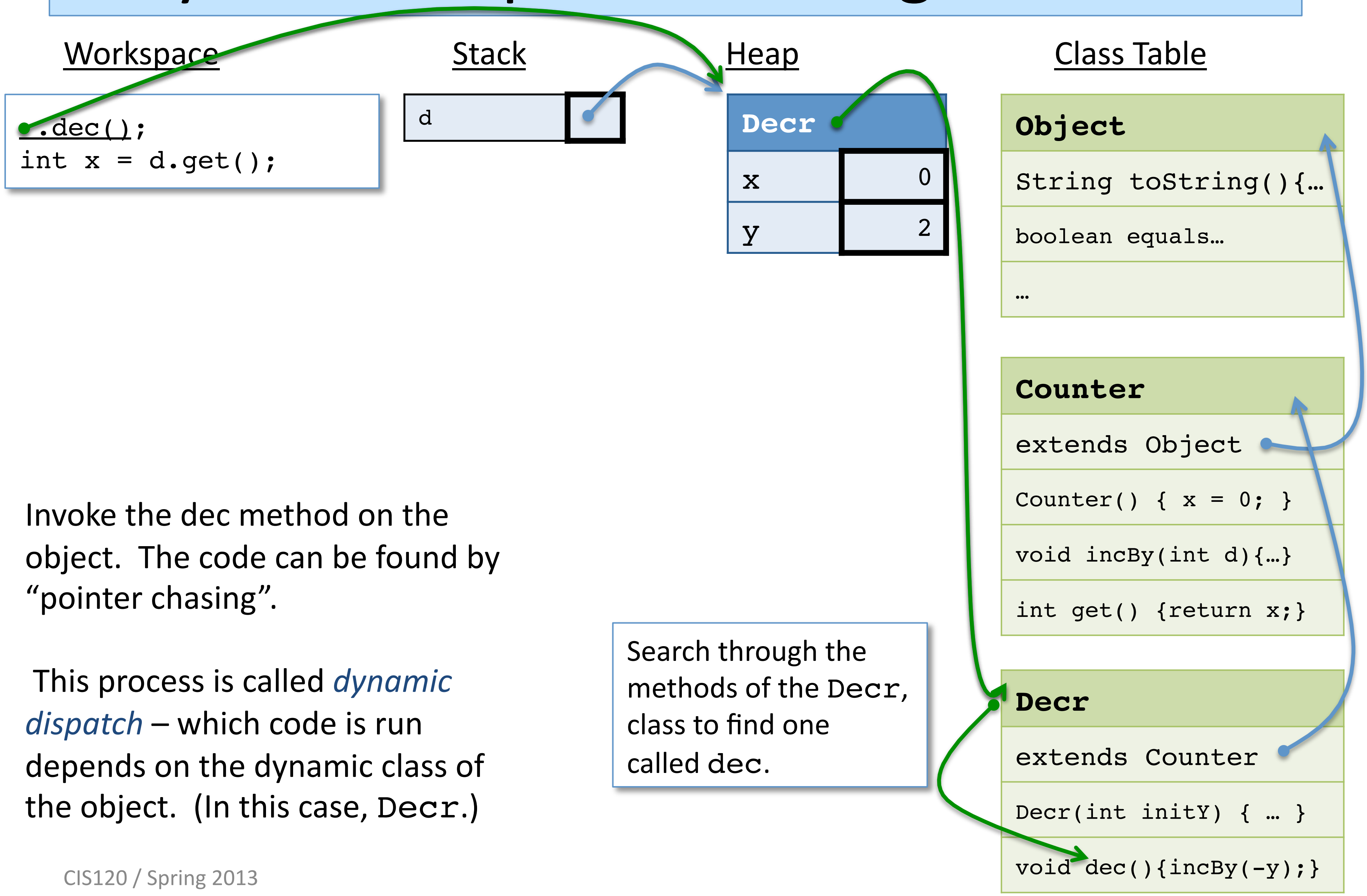
```
d.dec();
int x = d.get();
```



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



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 class of the object. (In this case, `Decr`.)

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

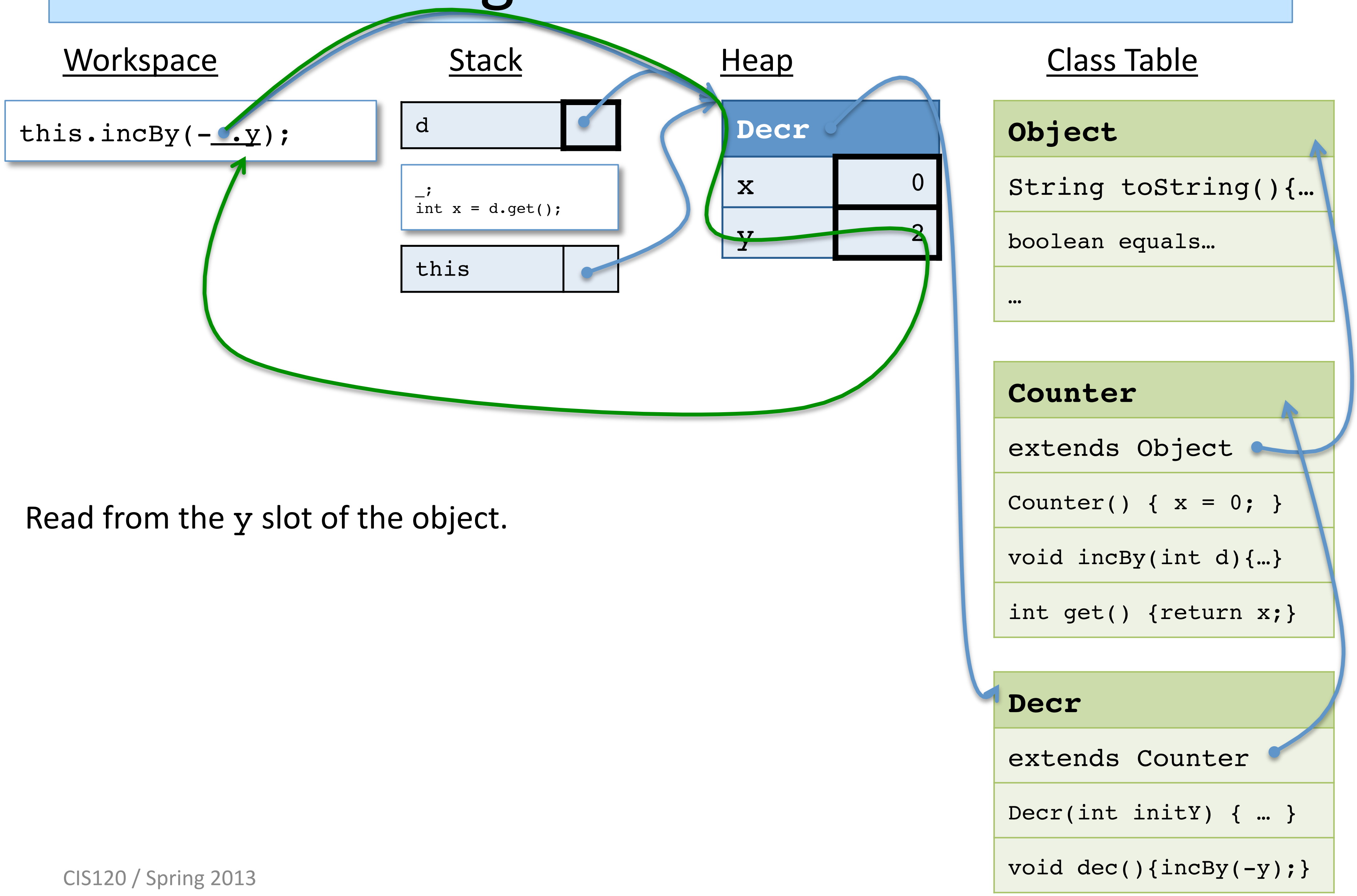
Object
String toString(){…
boolean equals…
…

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

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

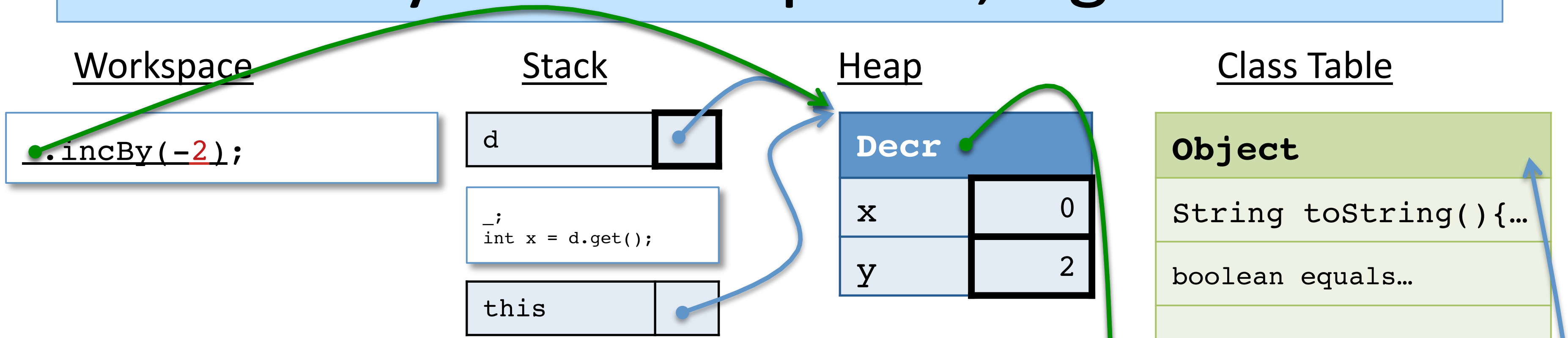
Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).

Reading A Field's Contents



Read from the y slot of the object.

Dynamic Dispatch, Again

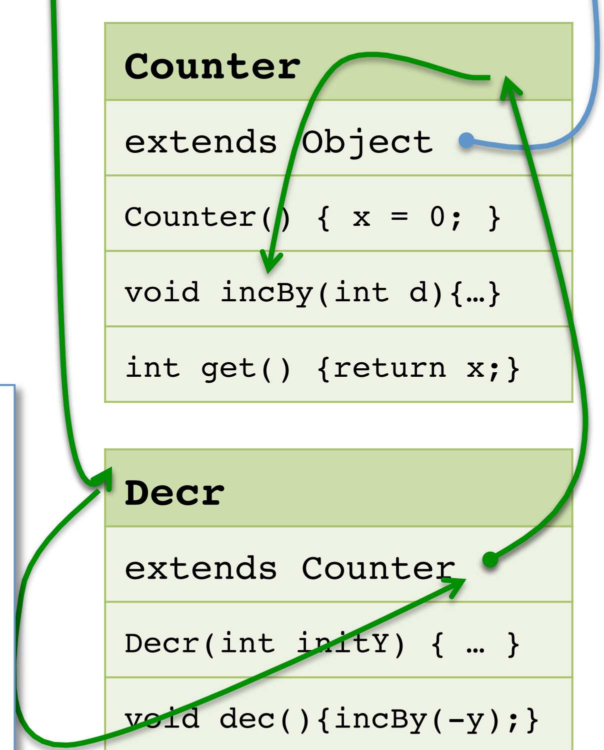


Invoke the `incBy` method on the object via dynamic dispatch.

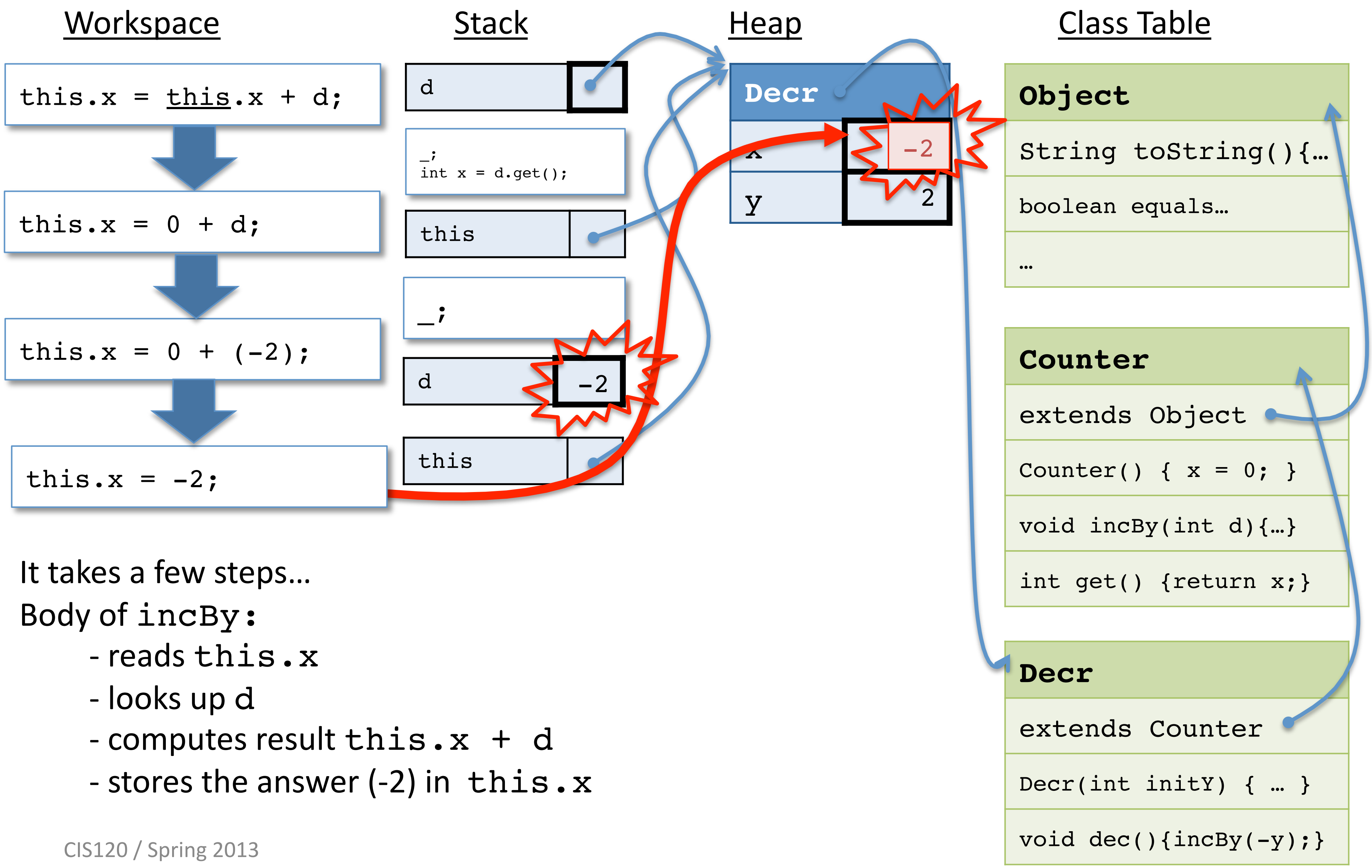
In this case, the `incBy` method is *inherited* from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.

The search is guaranteed to succeed – Java's static type system ensures this.

Search through the methods of the `Decr`, class to find one called `incBy`.
If the search fails, (recursively) search the parent class.



Running the body of incBy

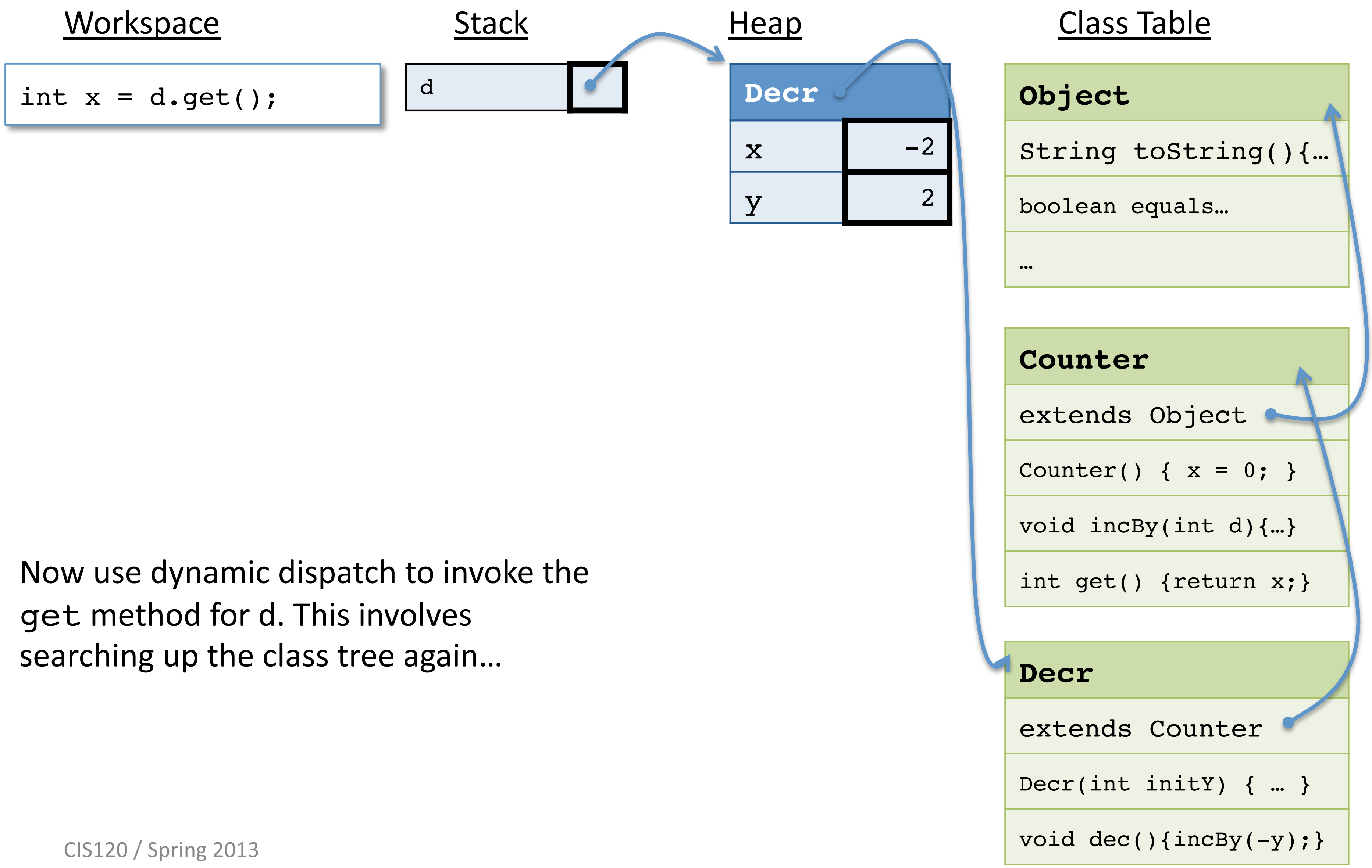


It takes a few steps…

Body of `incBy`:

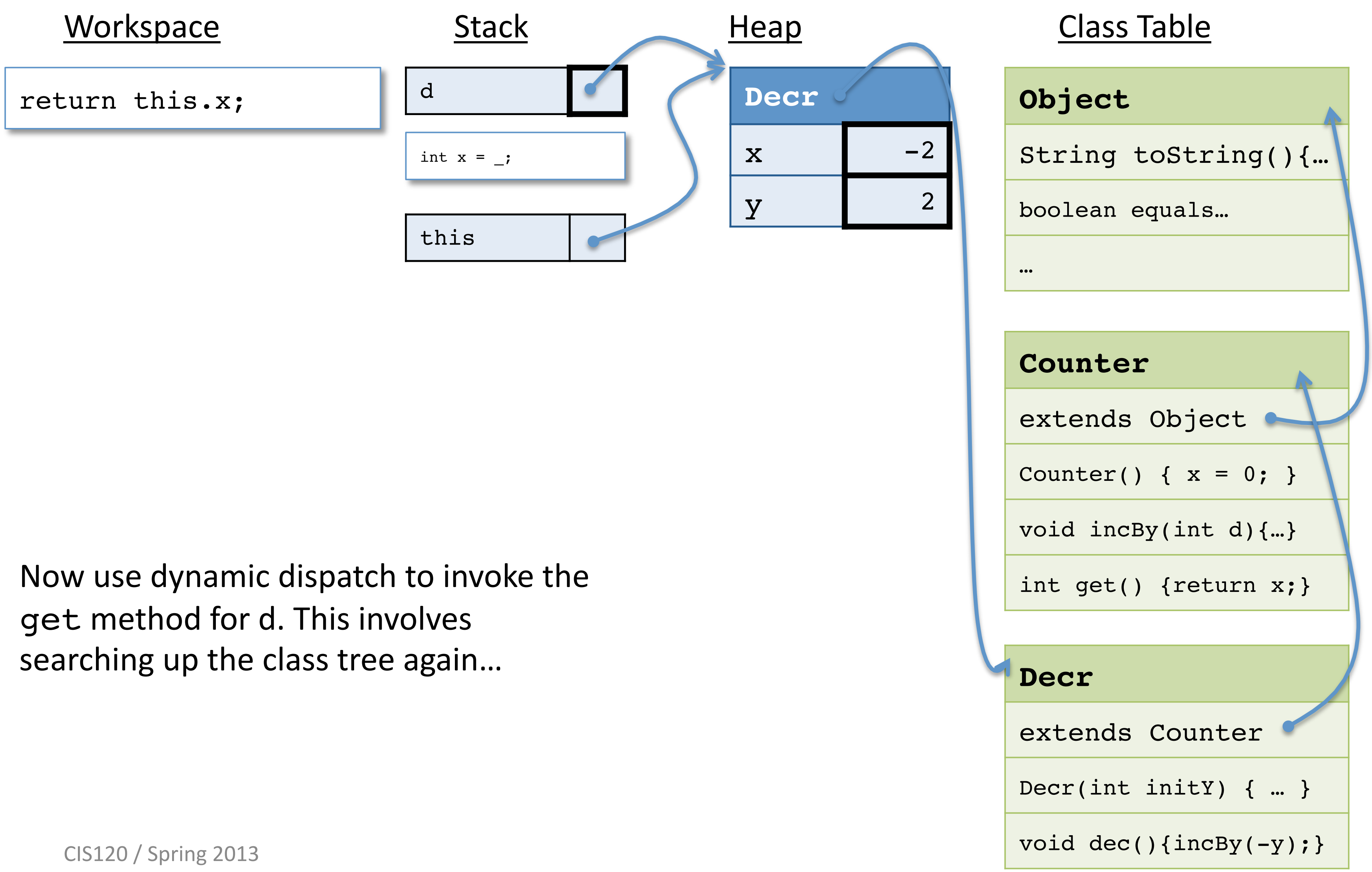
- reads `this.x`
- looks up `d`
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

After a few more steps…



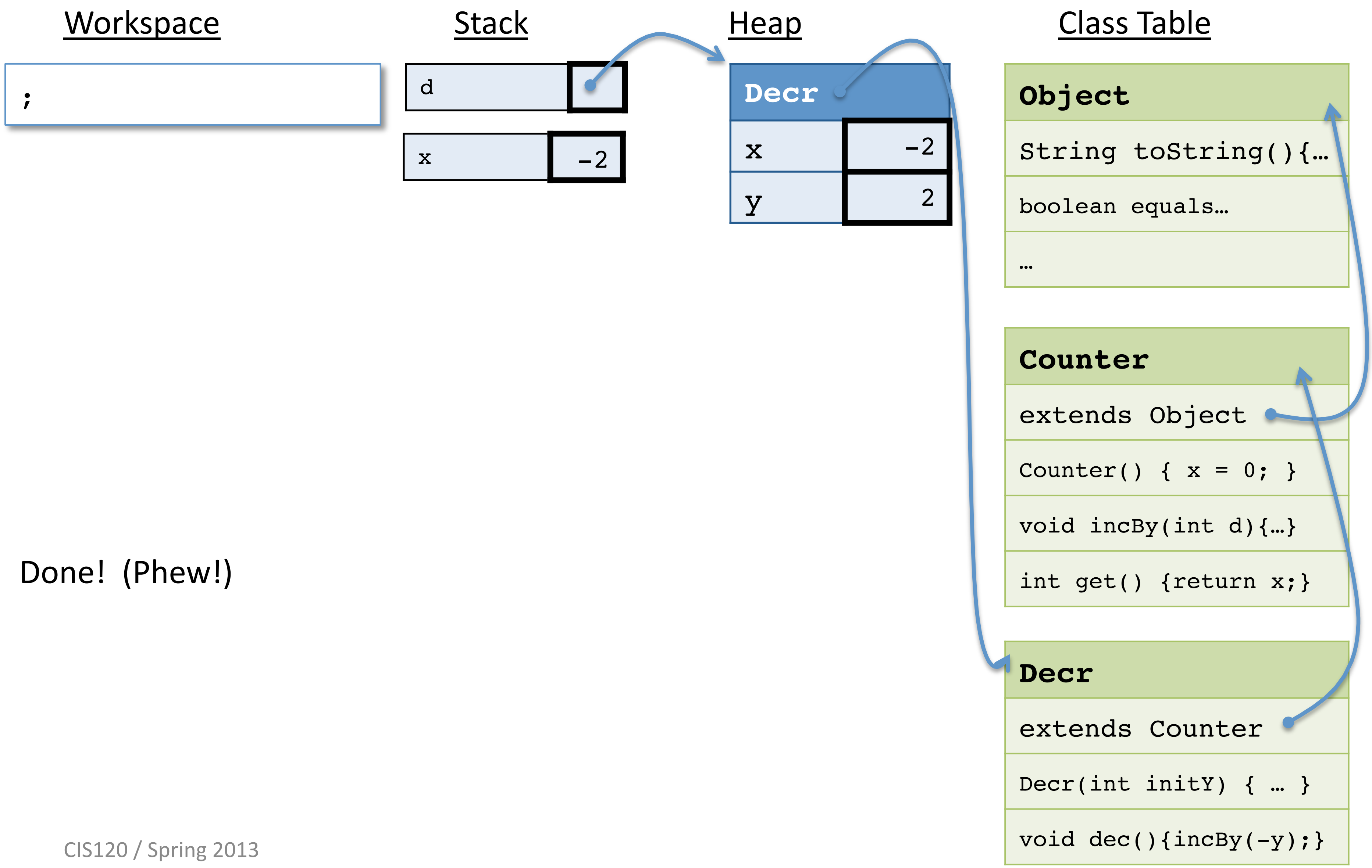
Now use dynamic dispatch to invoke the `get` method for d. This involves searching up the class tree again…

After a few more steps…



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* (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.

Java Generics

Queues in Java

Mutable Queue ML Interface

```
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

  (* Remove the first occurrence of the value. *)
  val remove : 'a -> 'a queue -> unit
end
```

Mutable Queue Java Interface

```
public interface Queue<E> {

    /** Determine if the queue is empty*/
    public boolean is_empty ();

    /** Add a value to the end of the queue */
    public void enq (E elt);

    /** Remove the front value and return it (if any) */
    public E deq ();

    /** Remove the first occurrence of the value */
    public void remove (E elt);

}
```

Interface comparison

```
module type QUEUE =
sig
  type 'a queue
  val create : unit -> 'a queue
  val is_empty :
      ‘a queue -> bool
  val enq :
      'a -> 'a queue -> unit
  val deq : 'a queue -> ‘a
  val remove :
      'a -> 'a queue -> unit
end
```

```
public interface Queue<E> {




    public boolean is_empty ();


    public void enq (E elt);


    public E deq ();
    public void remove (E elt);
}
```

Why Generics?

Subtype Polymorphism

```
public interface ObjQueue {
  public void enq(Object o);
  public Object deq();
  public boolean isEmpty();
  public boolean contains(Object o);
  …
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
___A___  x = q.deq();                // What type for A?  Object
System.out.println(x.trim());        // Is this valid?    No!
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();                // What type for B?  Object
```

Generics (Parametric Polymorphism)

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

```
Queue<String> q = …;

q.enq(" CIS 120 ");
___A___   x = q.deq();            // What type for A?   String
System.out.println(x.trim());     // Is this valid?     Yes!
q.enq(new Point(0.0,0.0));
___B___   y = q.deq();            // What type for B?   Point
```

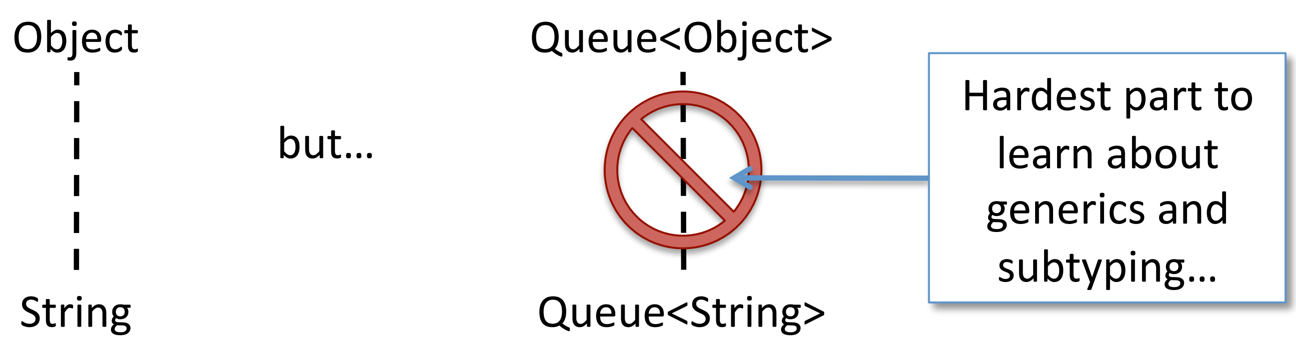
Subtyping and Generics

Subtyping and Generics

```
Queue<String> qs = new QueueImpl<String>();    // Ok?  Sure!
Queue<Object> qo = qs;                         // Ok?

qo.enq(new Object());                          Nononono!
String s = qs.deq(0);
```

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

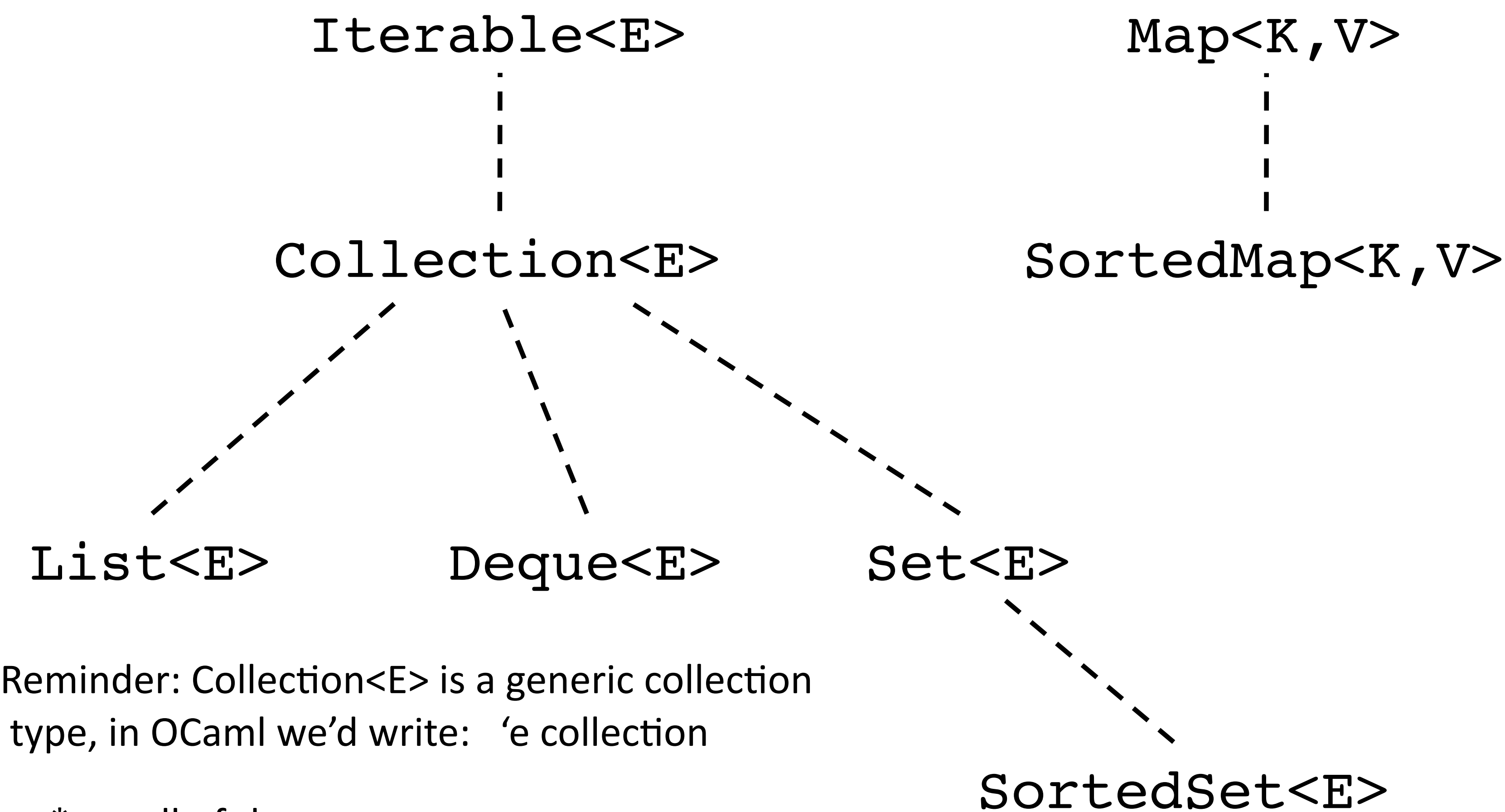


The Java Collections Library

A case study in subtyping and generics.

(Also very useful!)

Interfaces* of the Collections Library



Reminder: Collection<E> is a generic collection type, in OCaml we'd write: ‘e collection

*not all of them

Collection<E> Interface (Excerpt)

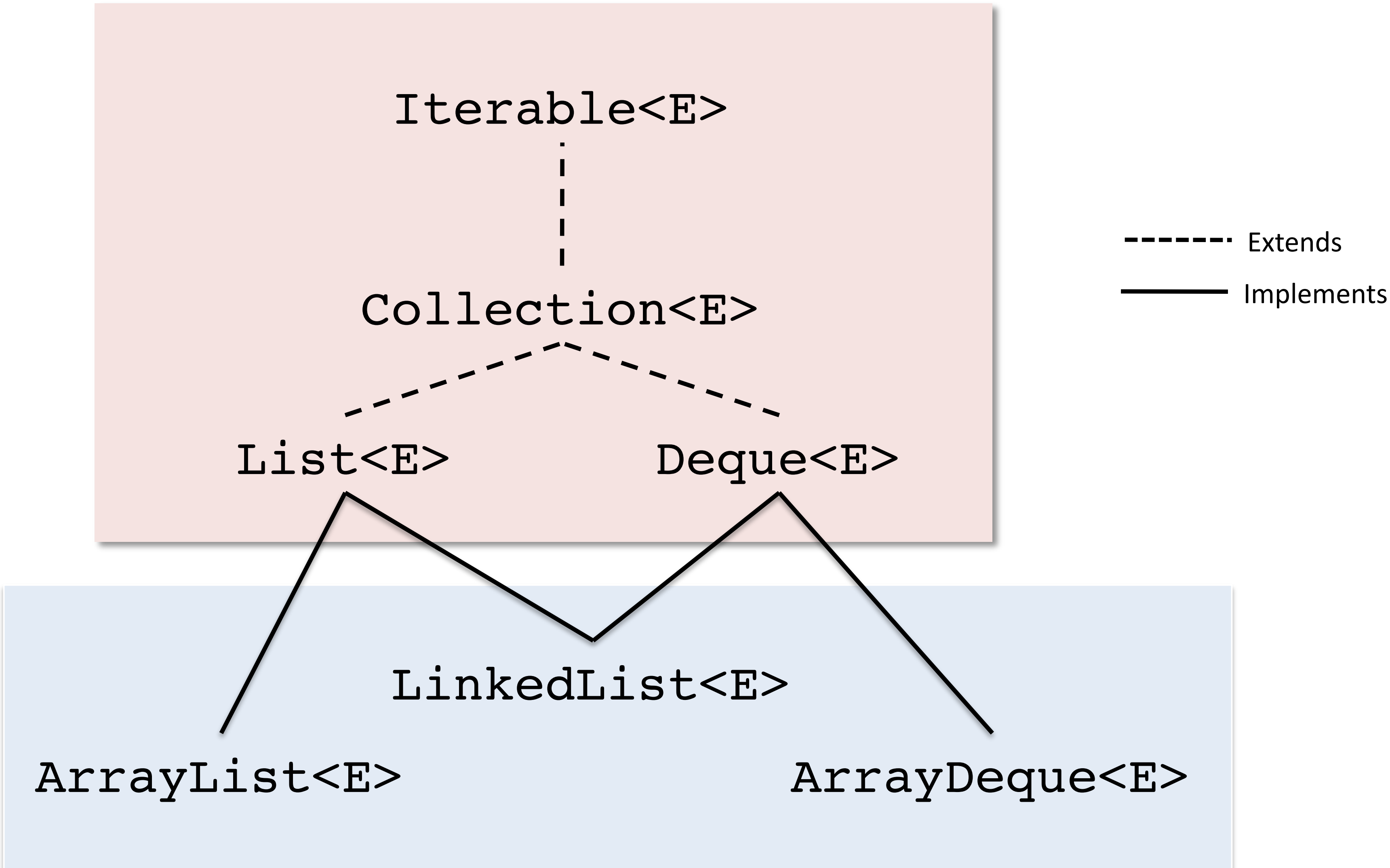
```
public interface Collection<E> extends Iterable<E> {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean remove(Object o);     // why not E?*
  boolean contains(Object o);

  // bulk operations
  …
}
```

- We've already seen this interface in the OCaml part of the course.
- Most collections are designed to be *mutable* (like queues)

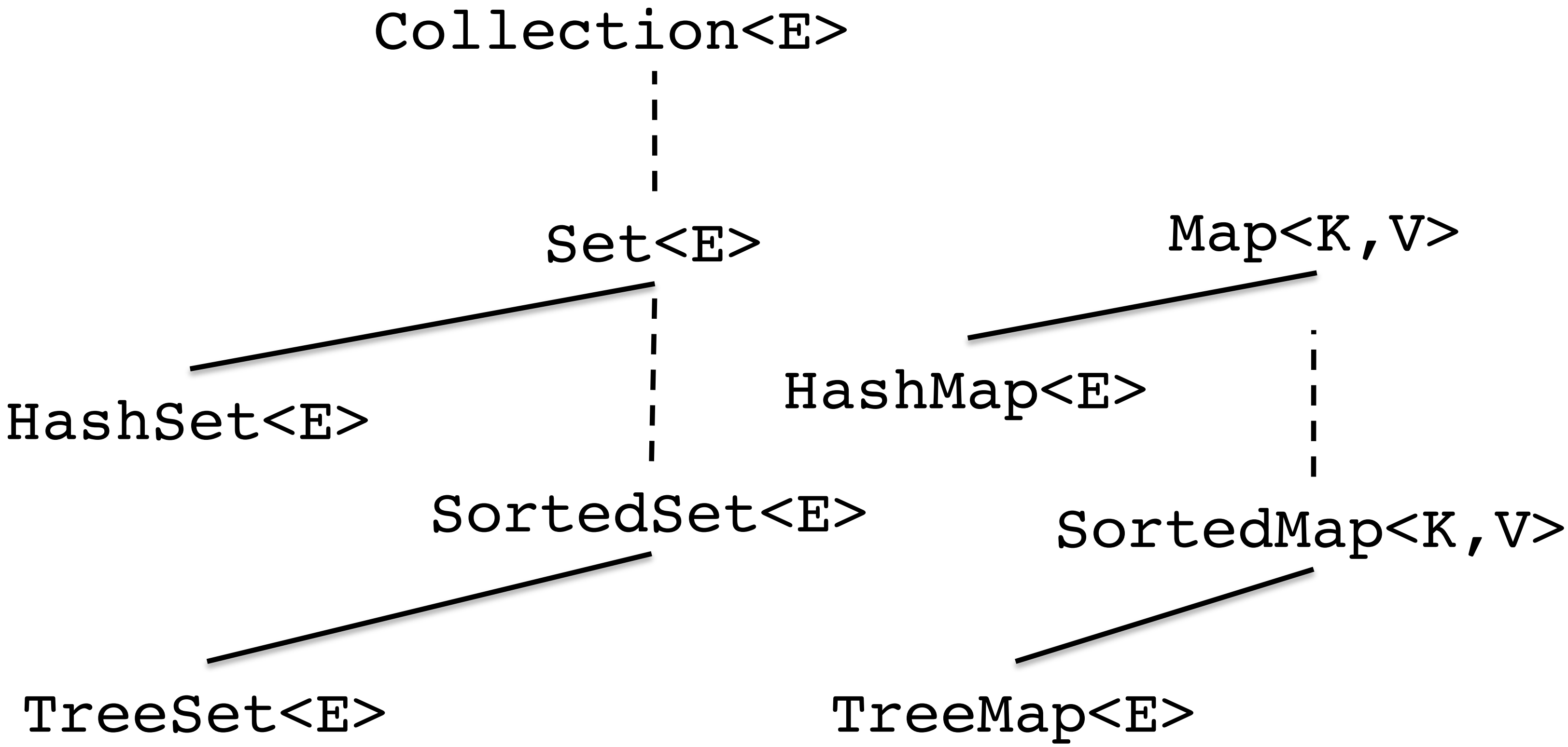
* Why not E? Internally, collections use the `equals` method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Sequences



Iterable<E>
Collection<E>
List<E>
Deque<E>
LinkedList<E>
ArrayList<E>
ArrayDeque<E>
Extends
Implements

Sets and Maps



Reading Java Docs

1. Collection<E>
2. List<E> and Set<E>
3. Iterable<E> and Iterator<E>

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

```
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete();   // optional
}
```

```
interface Iterable<E> {
    public Iterator<E> iterator();
}
```

While Loops

syntax:

```
// repeat body until condition becomes false
while (condition) {
   body
}
```

statement

boolean *guard* expression

example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

For Loops

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();
     book = iter.next()) {
   catalogue.addInfo(book);
   numBooks = numbooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

element type

array or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```

For-each Loops (Cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

Java Packages

- Java code can be organized into *packages* that provide namespace management.
  - Somewhat like OCaml's modules
  - Packages contain groups of related classes and interfaces.
  - Packages are organized hierarchically in a way that mimics the file system's directory structure.
- A .java file can *import* (parts of) packages that it needs access to:

```
import org.junit.Test;     // just the JUnit Test class
import java.util.*;        // everything in java.util
```

- Important packages:
  - java.lang , java.io , java.util , java.math, org.junit
- See documentation at: http://download.oracle.com/javase/6/docs/api/index.html