

Programming Languages and Techniques (CIS120)

Lecture 26

Static Types vs. Dynamic Classes,
The Java ASM, Java Generics

Chapter 24

Announcements

- Java Programming (Pennstagram)
 - Tuesday, November 5 at 11:59:59pm
- Upcoming: Midterm 2
 - Friday, November 8th in class
 - Coverage: mutable state, queues, dequeues, GUI, Java material up to TODAY (simple inheritance, "this")
- Exam Logistics:
 - Last Names A – M go to Leidy Labs 10 ([here](#))
 - Last Names N – Z go to College Hall 200 (COLL 200)
- Midterm Review Session:
 - Wednesday, November 6th 6:00-8:00pm in Towne 100
 - RSVP on Piazza

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);  
}
```

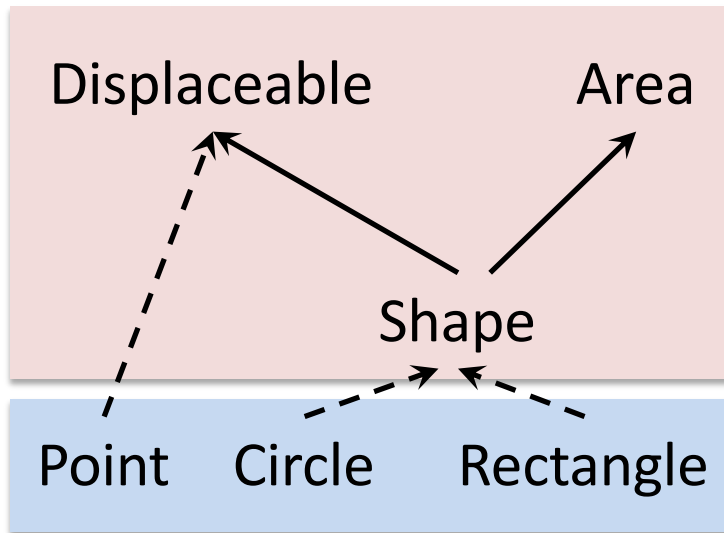
```
public interface Area {  
    double getArea();  
}
```

```
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.

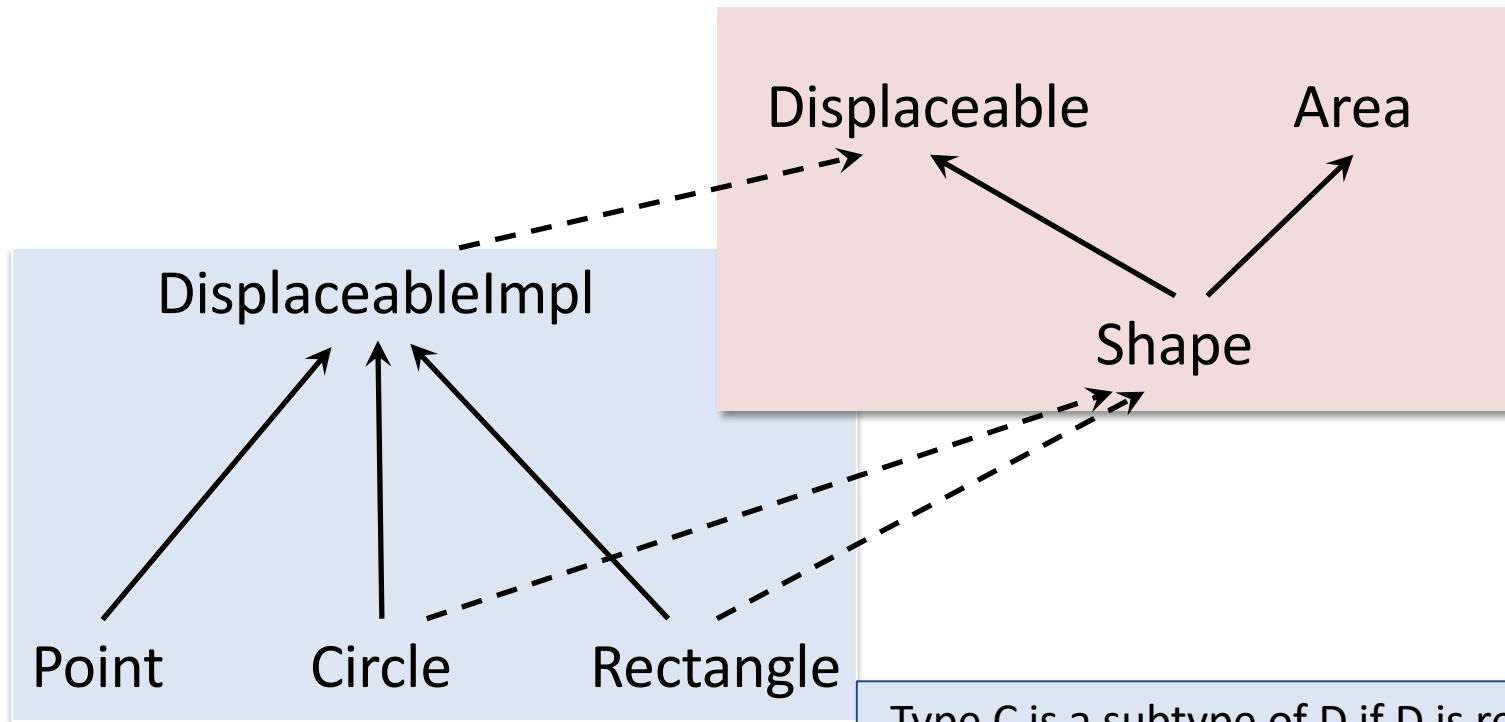
Class Extension: Inheritance

- 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*, the subclass only *adds* new fields or methods.
- Use simple inheritance to *share common code* among related classes.
- Example: Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

Subtyping with Inheritance



- Extends
- - - Implements

-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

Example of Simple Inheritance

See: [Shapes.zip](#)

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)

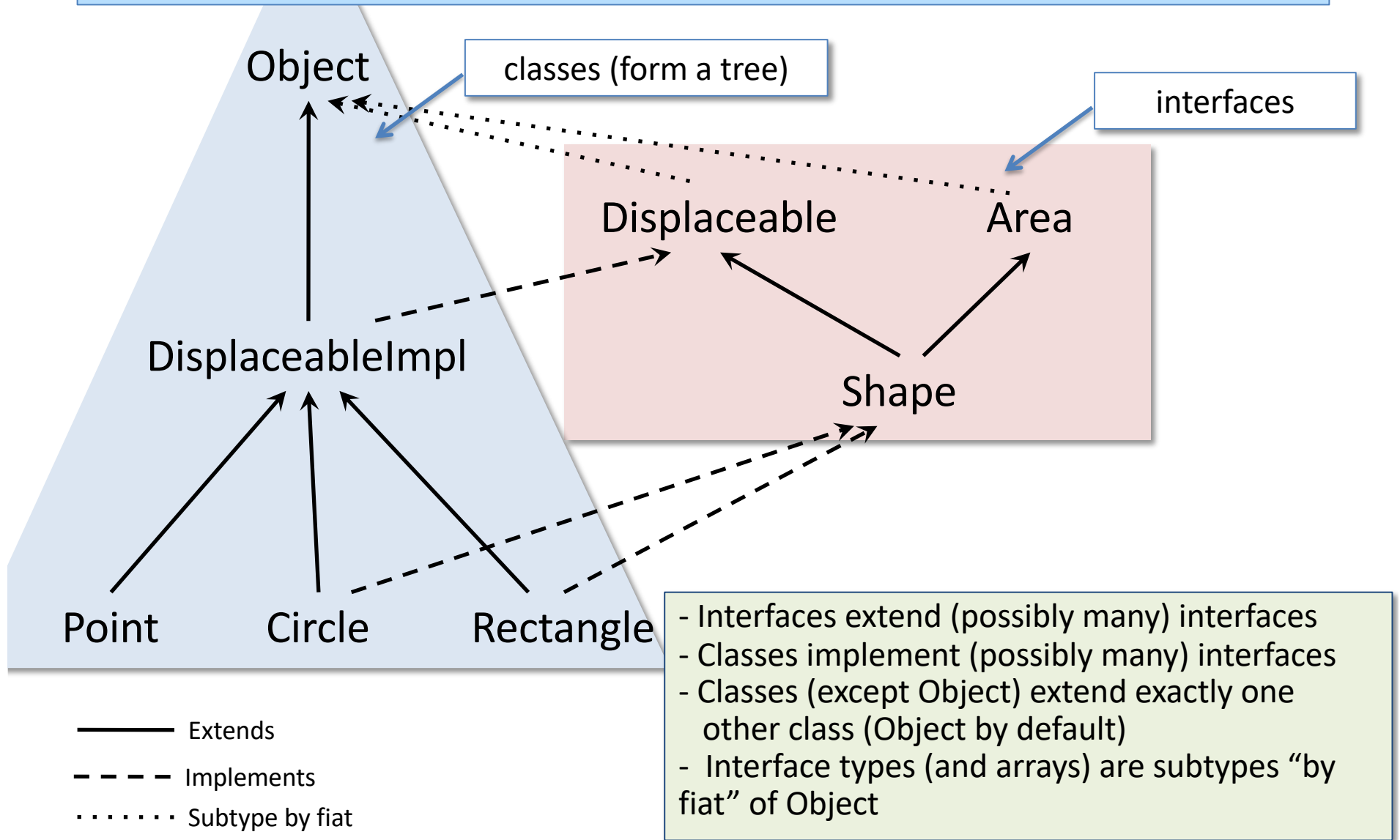
```
public Circle(Point pt, int radius) {  
    super(pt.getX(),pt.getY());  
    this.radius = radius;  
}
```

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



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 tricky to use properly, and the need for them arises only in somewhat special cases
 - Designing complex, reusable libraries
 - Special 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;
```

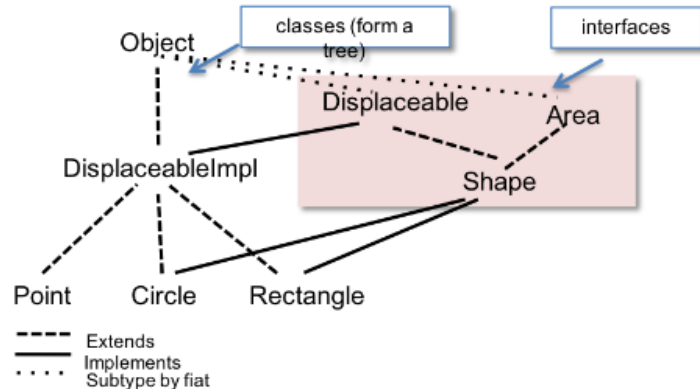
- The **dynamic class** of an *object* is the class that it was created from at run time

```
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

What is the static type of *a1* on line A?

```
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);
```



Area

Circle

None of
the above

Not well
typed

Static type vs. Dynamic type

```
public Area asArea (Area a)
    { return a; }
```

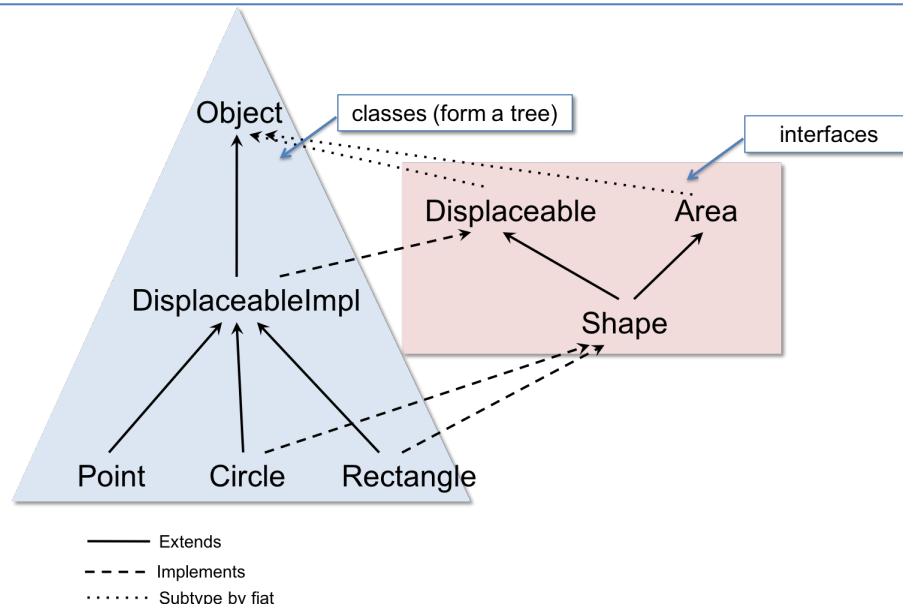
...

```
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?

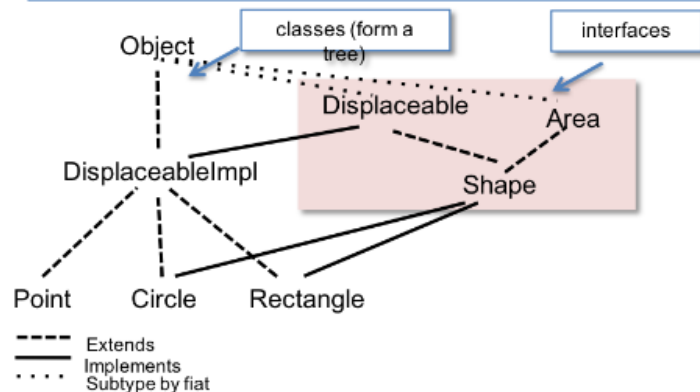
1. Area
2. Circle
3. None of the above
4. Not well typed



Area

What is the dynamic class of *a1* when execution reaches A?

```
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);
```



Area

Circle

None of the above

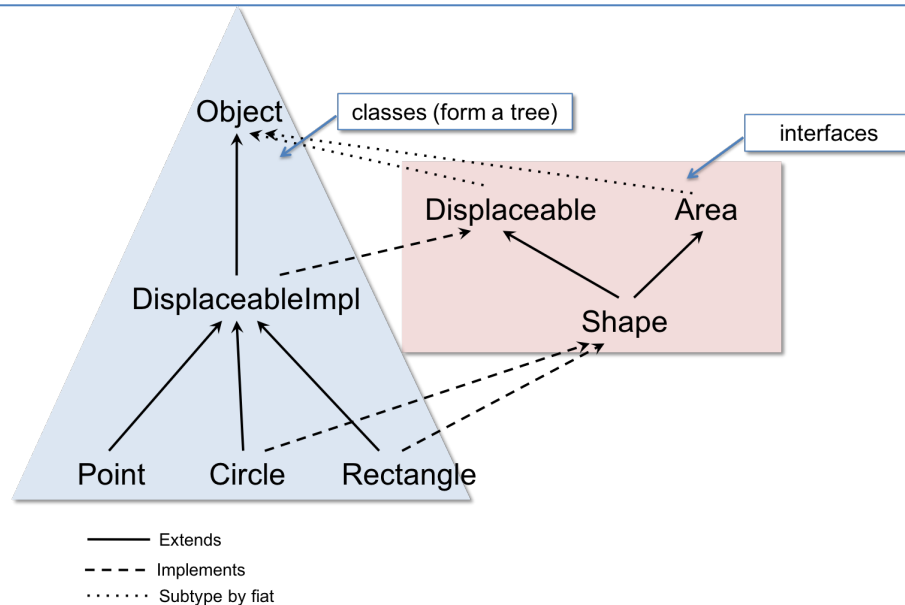
Not well typed

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?

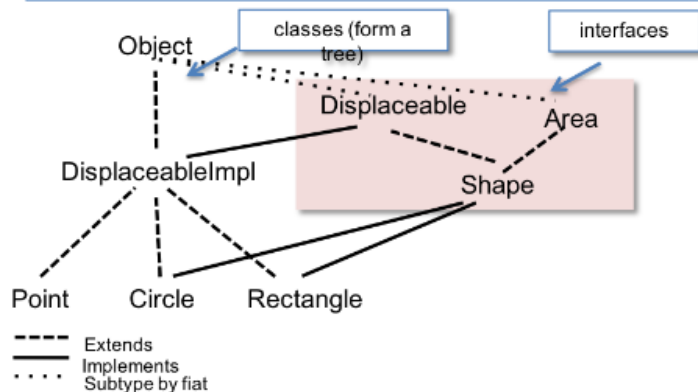
1. Area
2. Circle
3. None of the above
4. Not well typed



Circle

What type could we declare for *x* (in blank B)?

```
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);
```



- Area
- Circle
- None of the above
- Not well typed

Static type vs. Dynamic type

```
public Area asArea (Area a)
    { return a; }
```

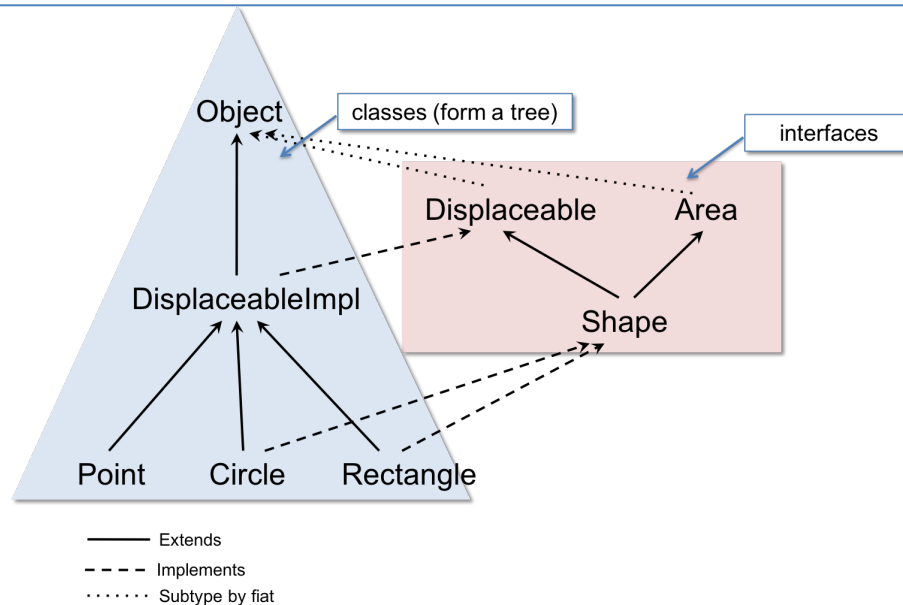
...

```
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'?

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; }
}

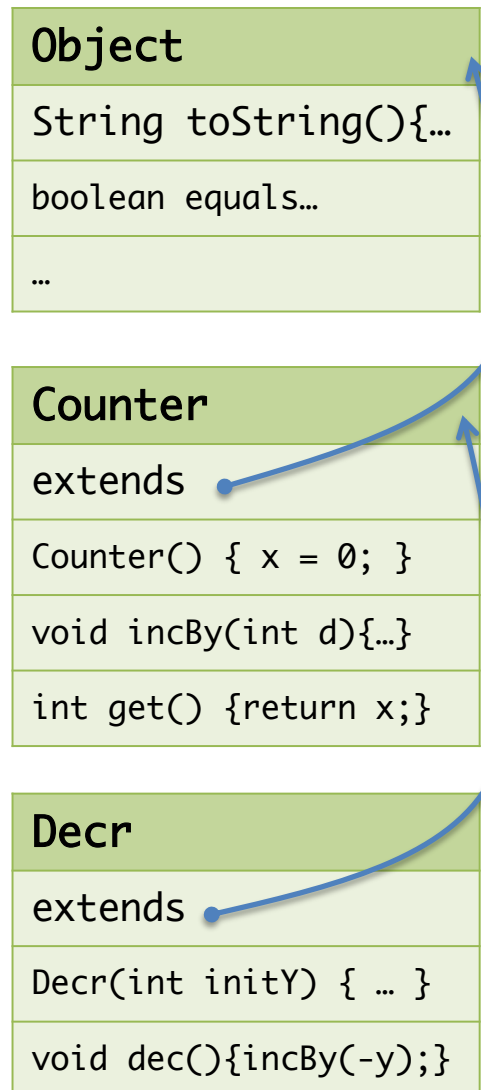
public class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}
```

The class table contains:

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

Class Table

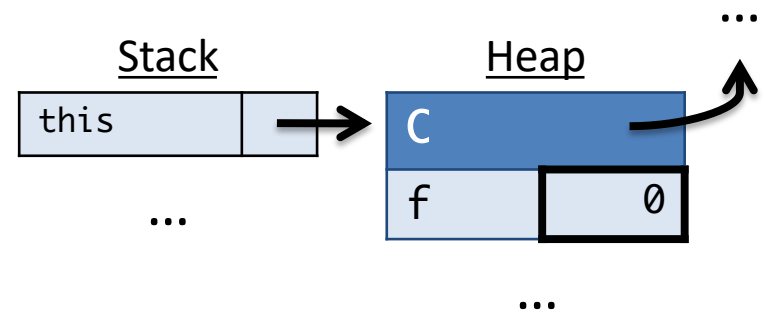
Object
String toString(){...}
boolean equals...
...
Counter
extends 
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}
Decr
extends 
Decr(int initY) { ... }
void dec(){incBy(-y);}



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

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

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

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;}
```

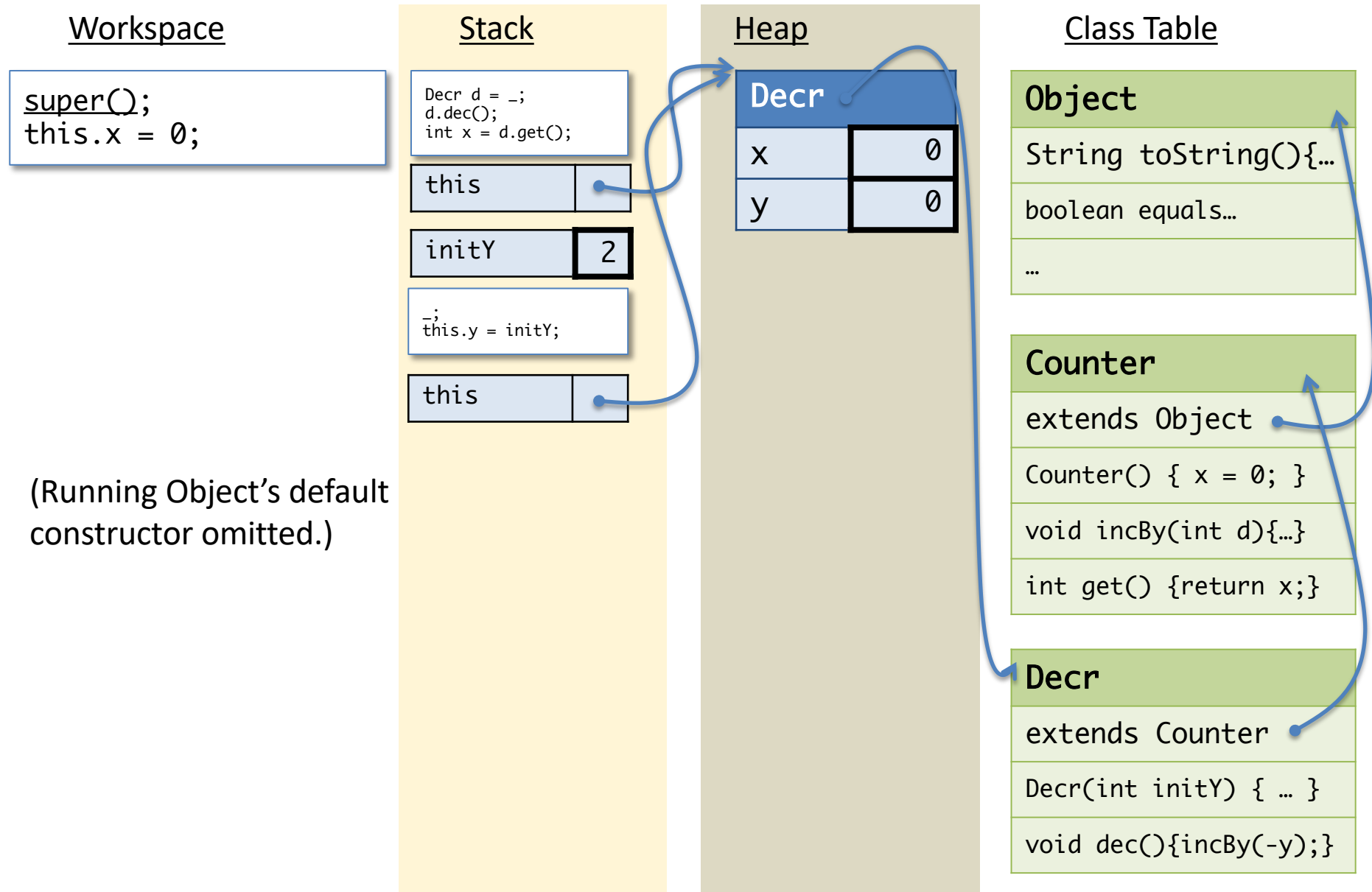
Decr

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

Call to `super`:

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

Abstract Stack Machine



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;}
```

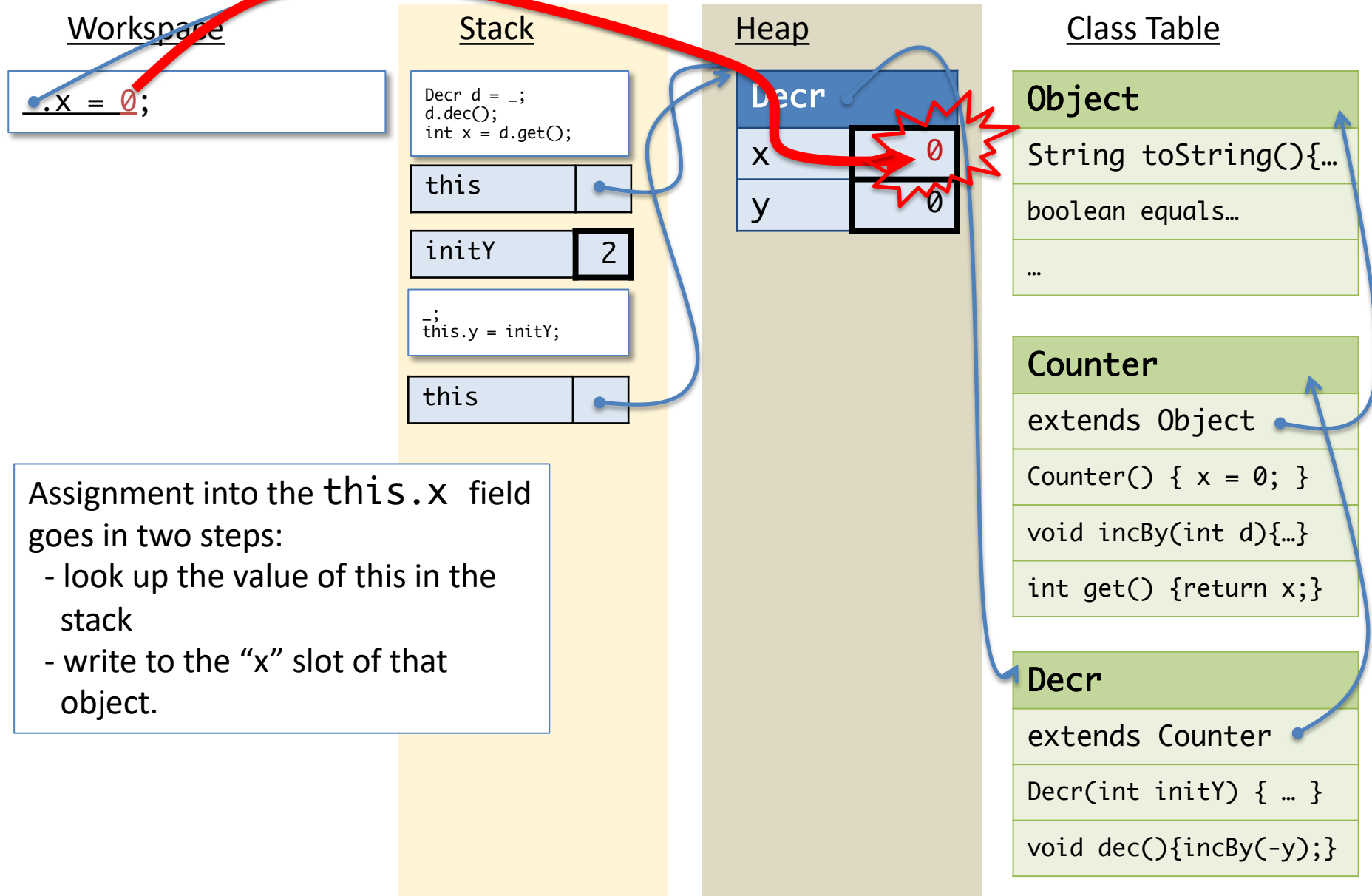
Decr

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

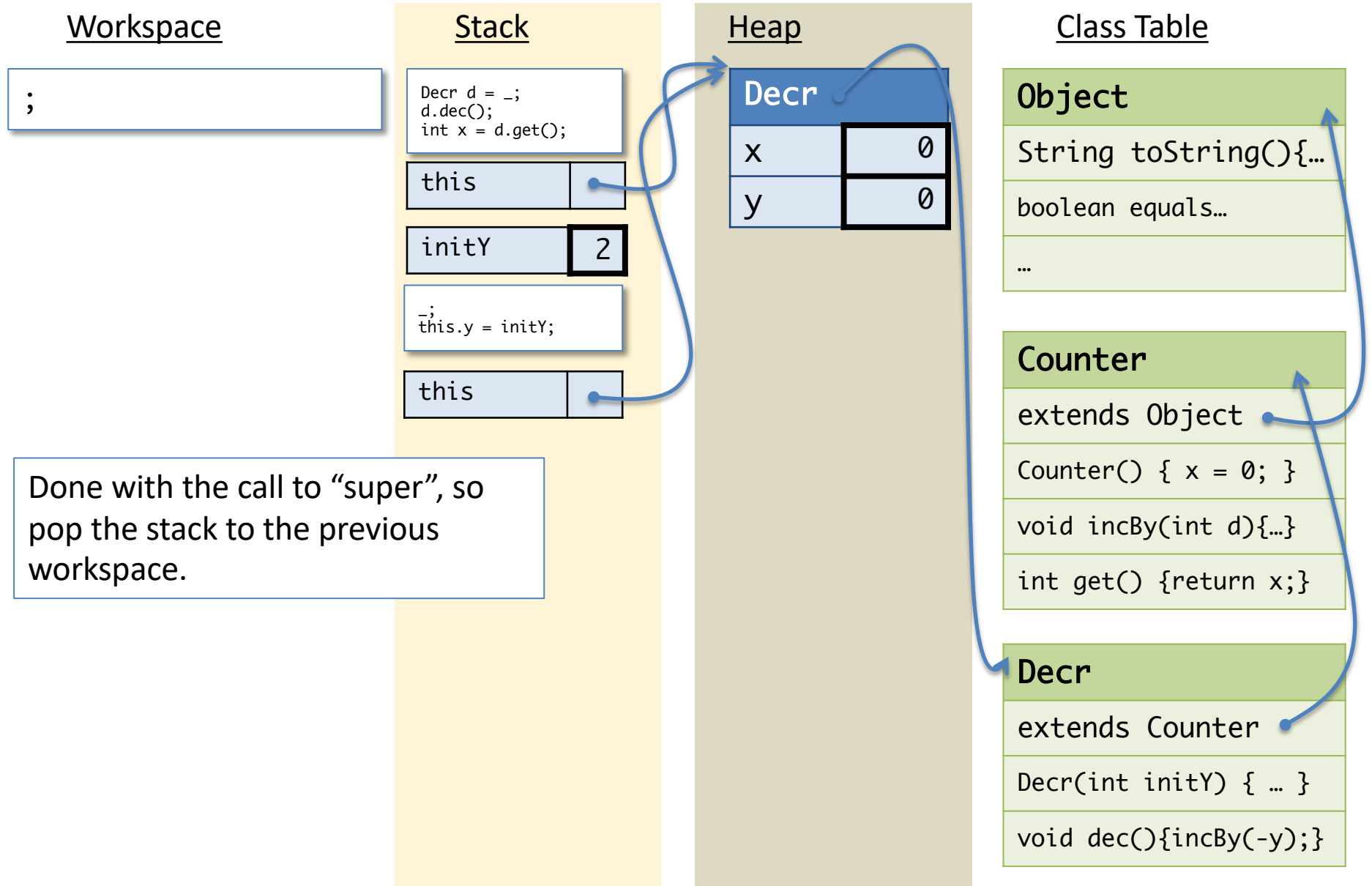
Assignment into the `this.x` field goes in two steps:

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

Assigning to a Field



Done with the call



Continuing

Workspace

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

Continue in the Decr class's constructor.

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;}
```

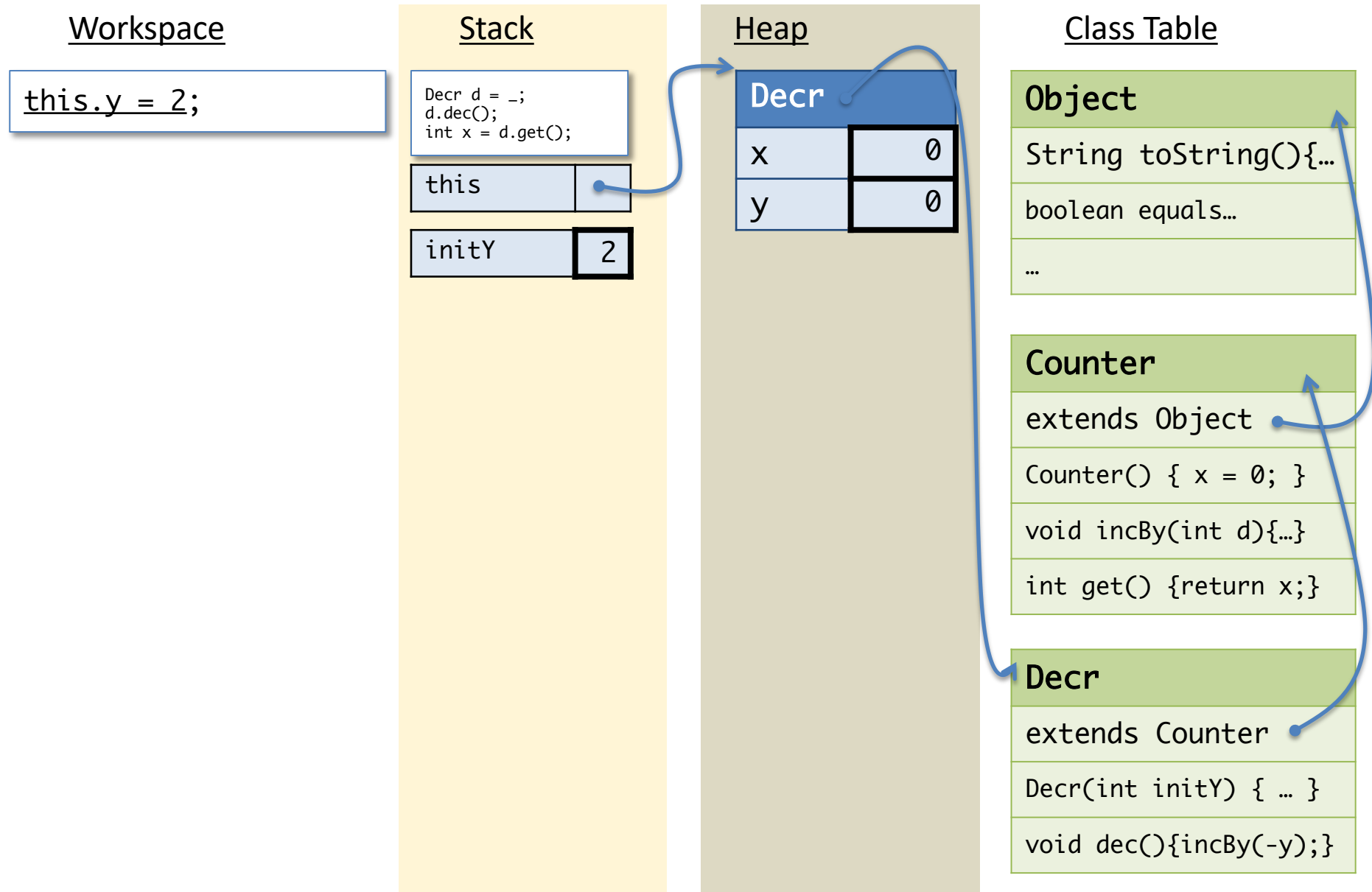
Decr

```
extends Counter
```

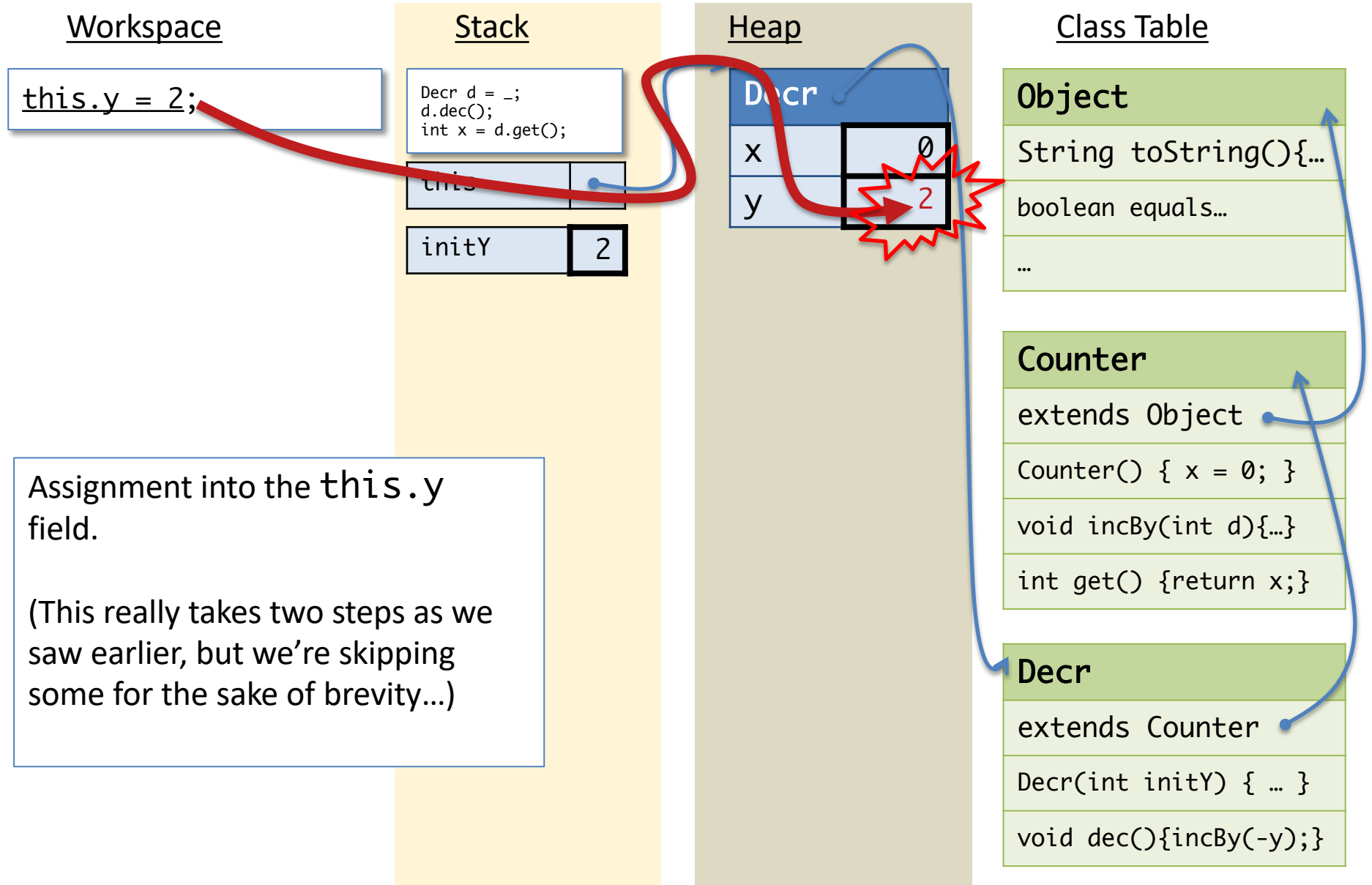
```
Decr(int initY) { ... }
```

```
void dec(){incBy(-y);}
```

Abstract Stack Machine



Assigning to a field



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

Workspace

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

Stack

Heap

Decr	
x	0
y	2

Class Table

Object

```
String toString(){...}  
boolean equals...  
...
```

Counter

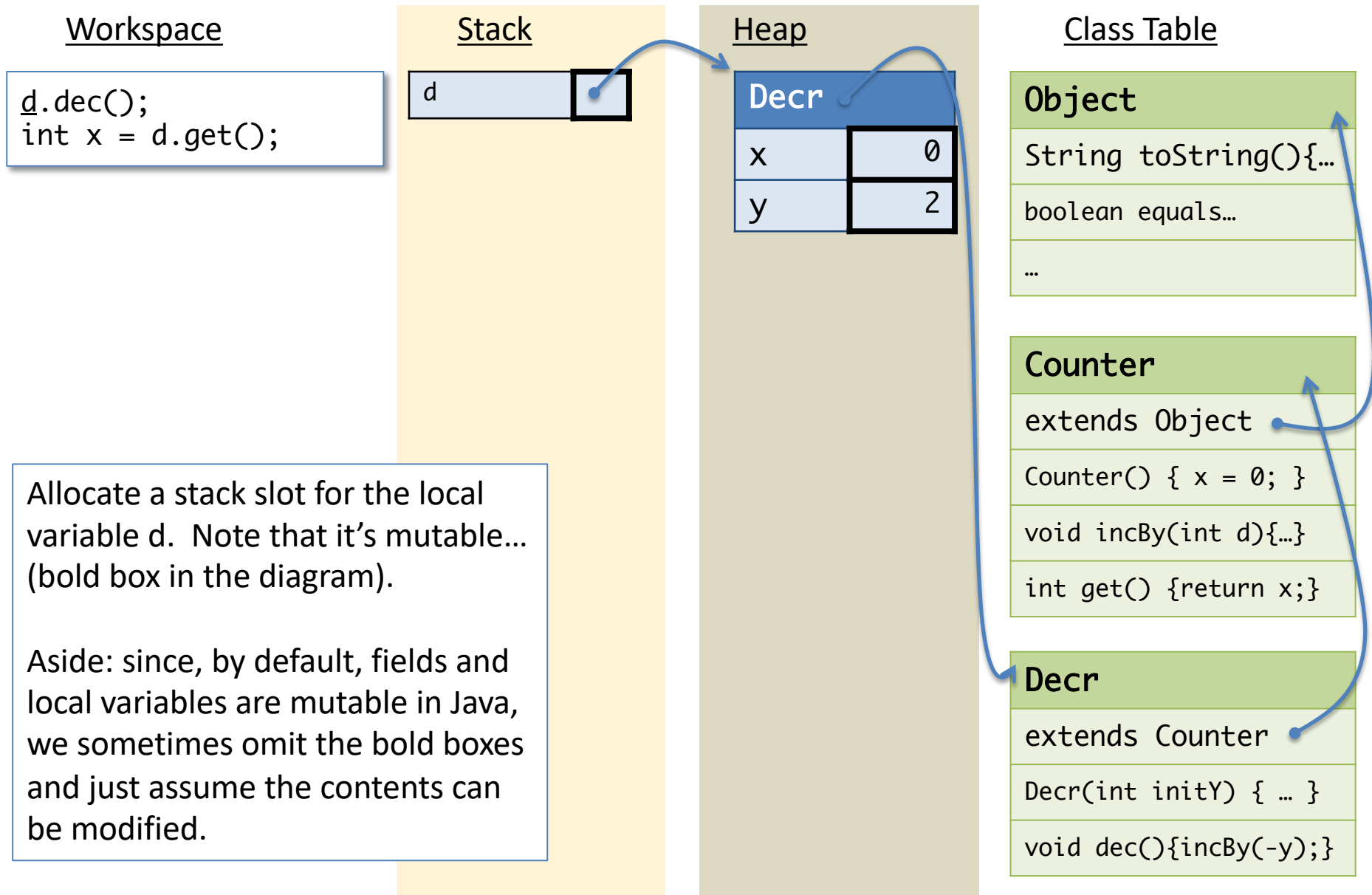
```
extends Object  
Counter() { x = 0; }  
void incBy(int d){...}  
int get() {return x;}
```

Decr

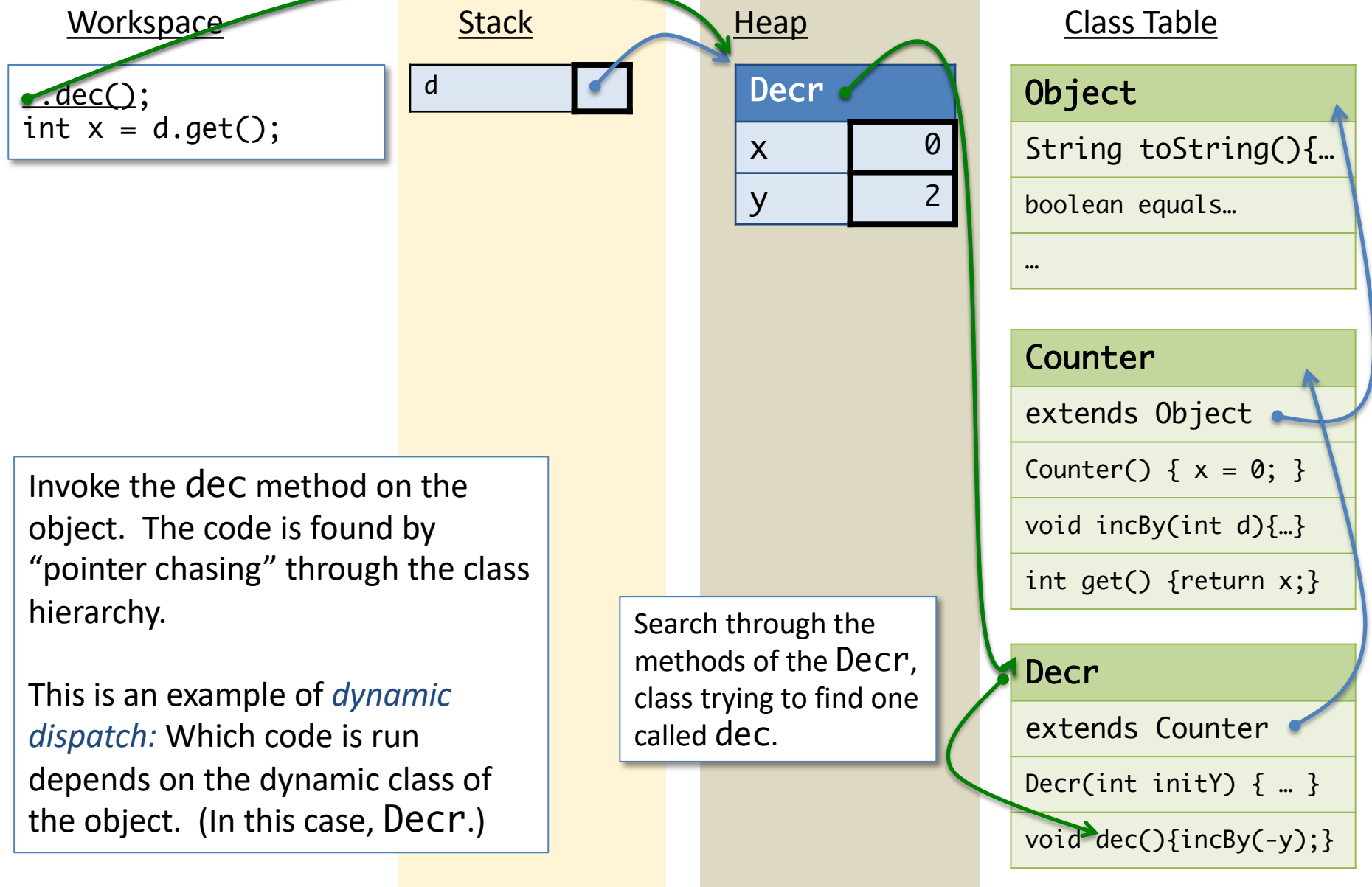
```
extends Counter  
Decr(int initY) { ... }  
void dec(){incBy(-y);}
```

Continue executing the program.

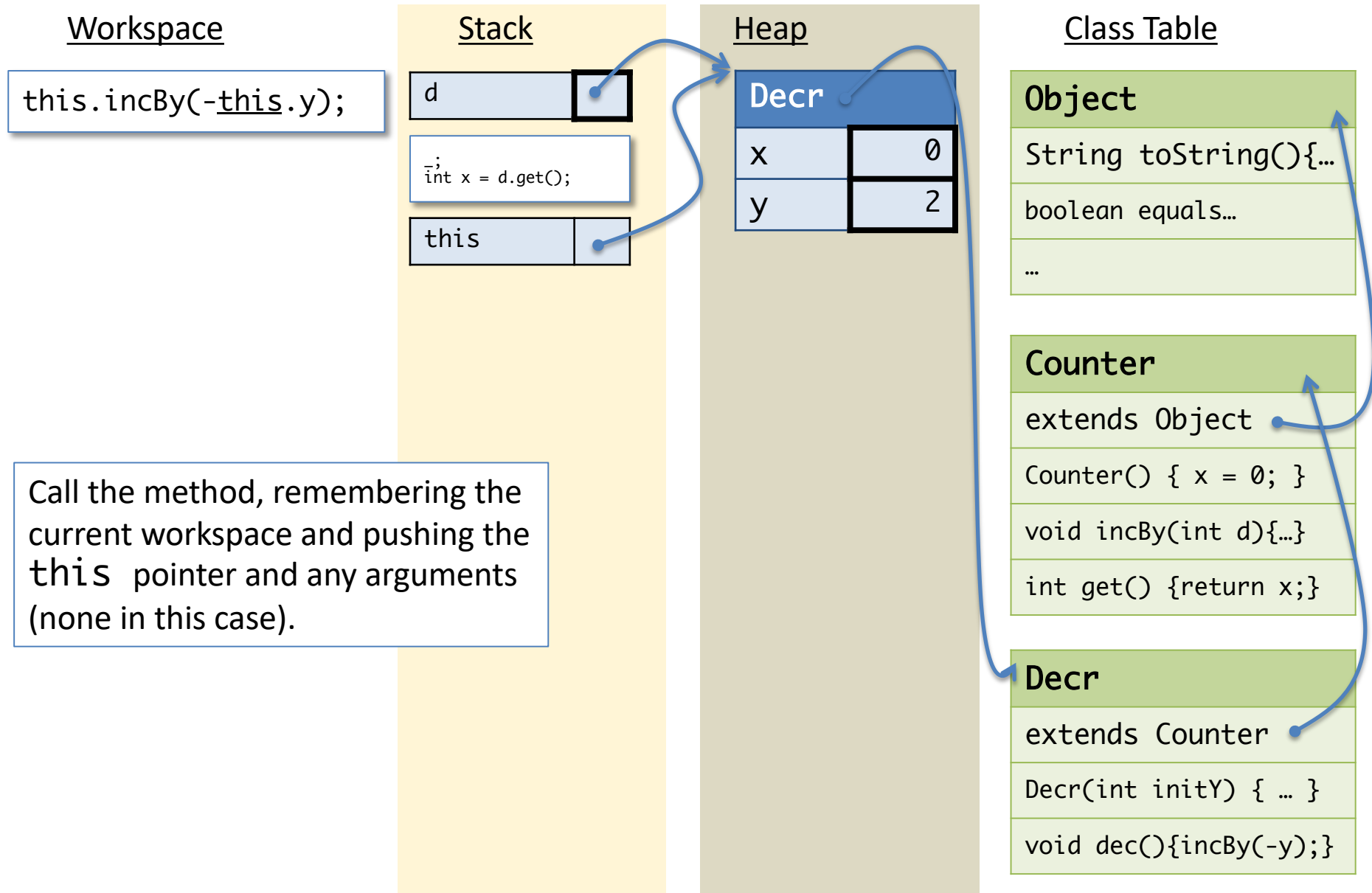
Allocating a local variable



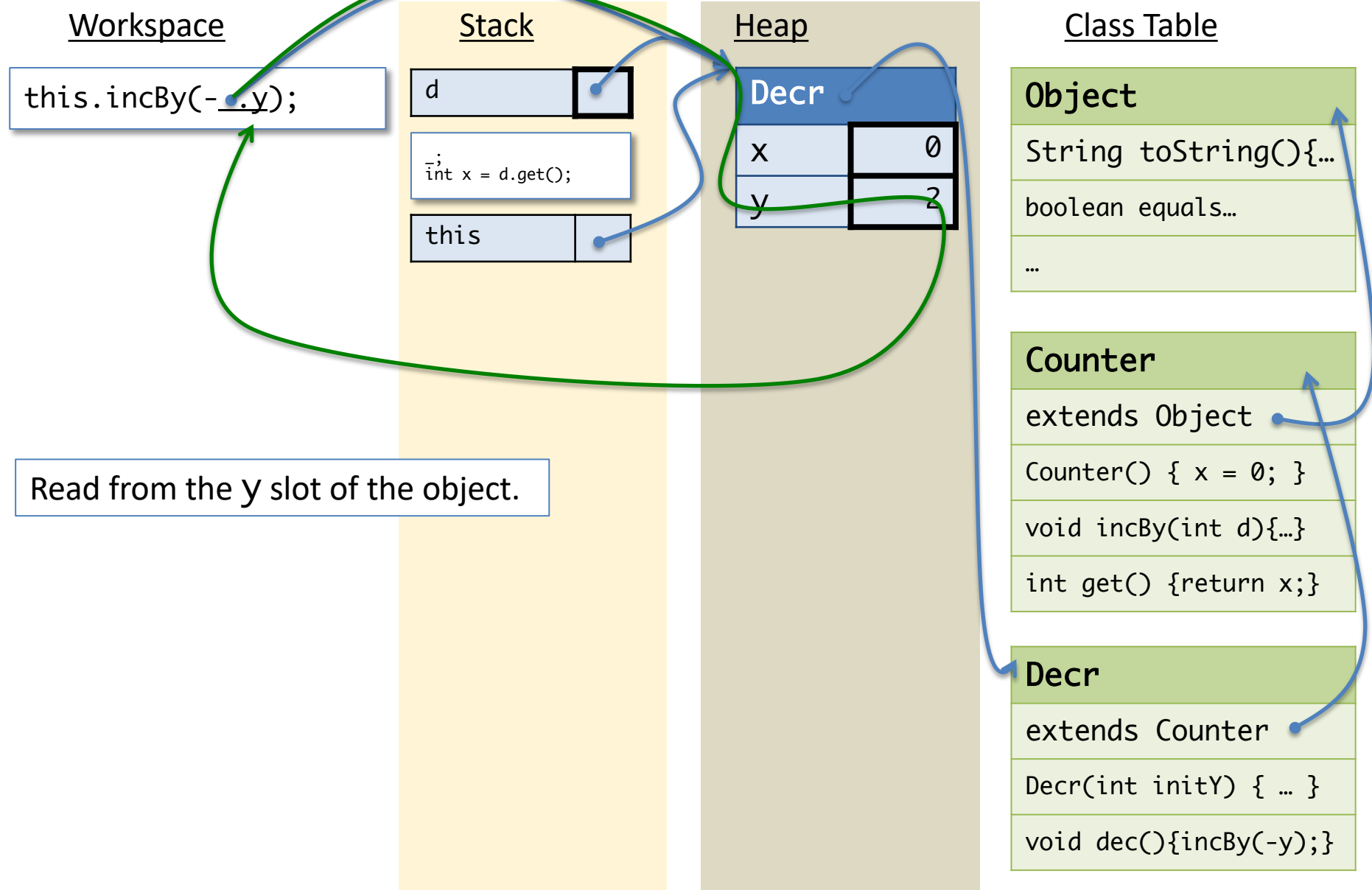
Dynamic Dispatch: Finding the Code



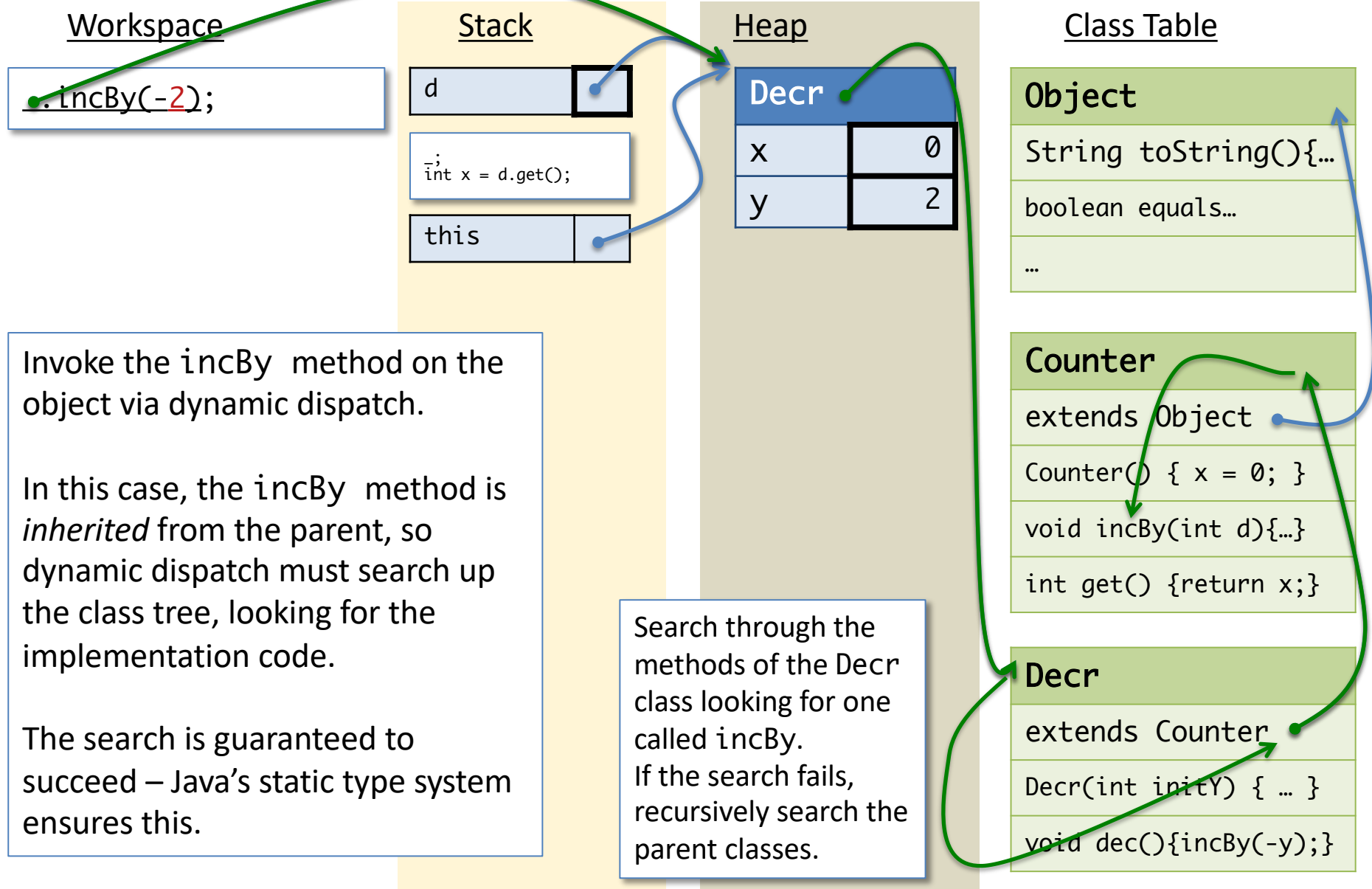
Dynamic Dispatch: Finding the Code



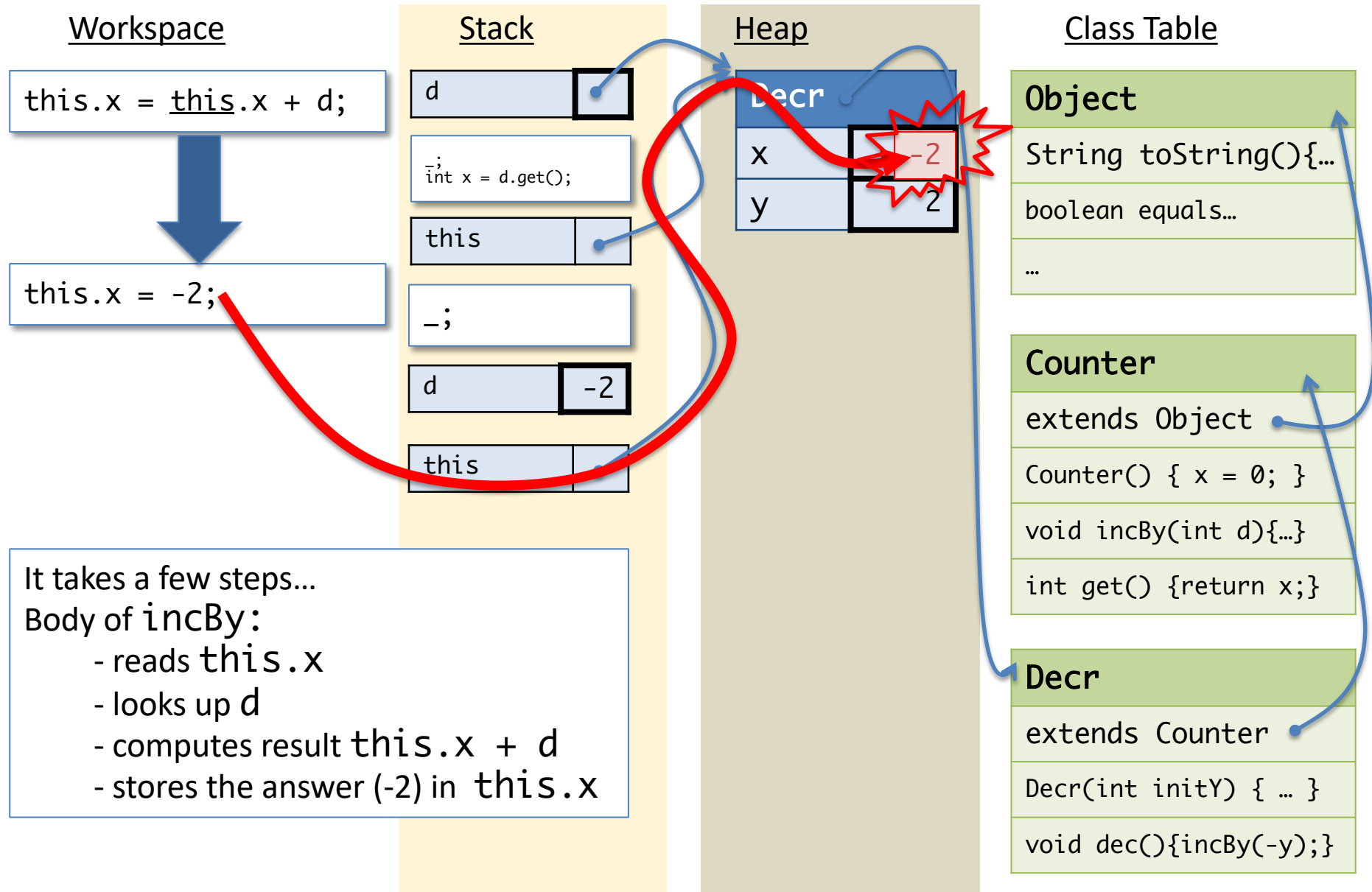
Reading a Field's Contents



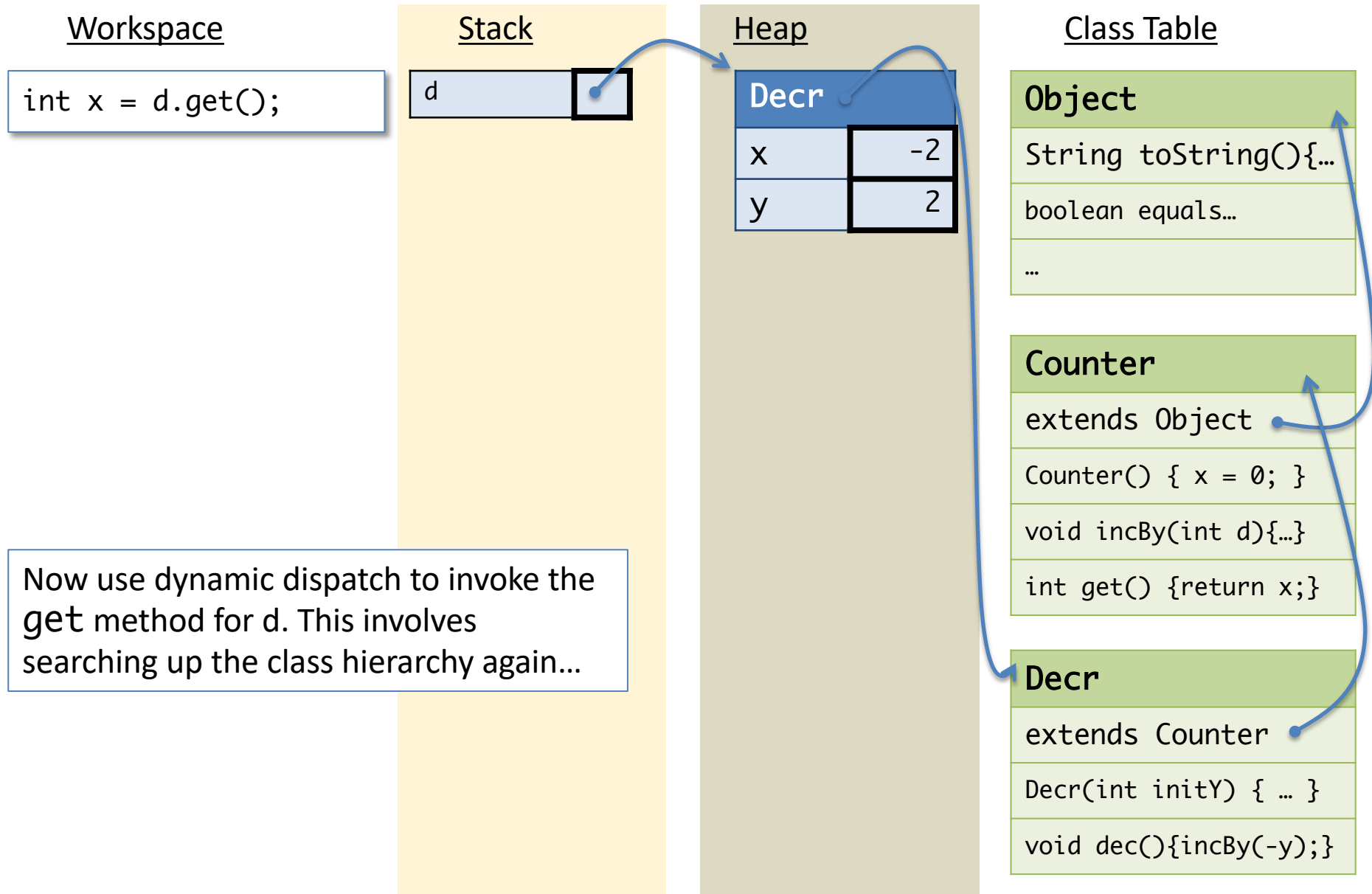
Dynamic Dispatch, Again



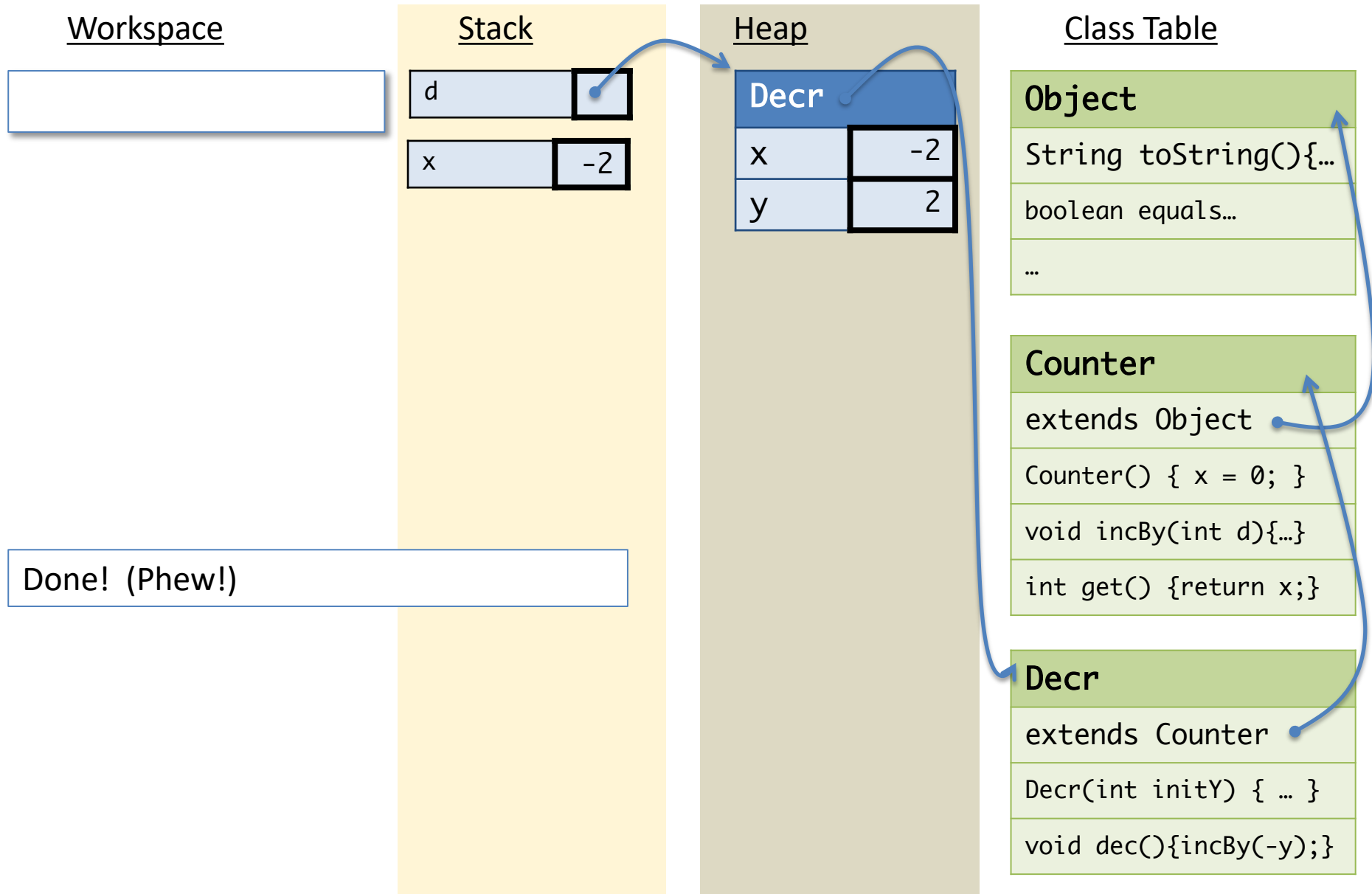
Running the body of incBy



After 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.

What is the value of x at the end of this computation?

```
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();
```

-2

-1

0

1

2

NullPointerException

Doesn't type check

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

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()`

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` pointer
 - 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

```
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

```
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> {  
    public void enq(E o);  
    public E deq();  
    public 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> {  
    public void enq(E o);  
    public E deq();  
    public boolean isEmpty();  
    ...  
}
```

```
Queue<String> q = ...;
```

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

```
// What type of x?   String  
// Is this valid?   Yes!  
// Is this valid?   No!
```

Subtyping and Generics

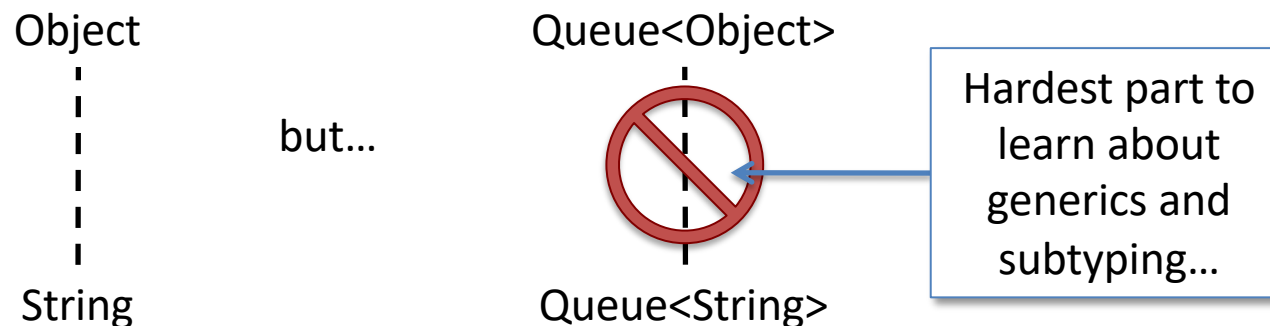
Subtyping and Generics*

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

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

Other subtleties with Generics

- Unlike OCaml, Java classes and methods can be generic only with respect to *reference* types.
 - Not possible to do: `Queue<int>`
 - Must instead do: `Queue<Integer>`
- Java Arrays cannot be generic
 - Not possible to do:

```
class C<E> {  
    E[] genericArray;  
    public C() {  
        genericArray = new E[];  
    }  
}
```

