Warmup

```
int a = ???;
double b = ???;
double c = ???;
return (b*b > 4*a*c) ?
       2*a : 2*c;
```

What are the static types of these (sub)expressions?

1. `4*a`
2. `4*a*c`
3. `b*b > 4*a*c`
4. The returned value

Warmup 1

```
int a = ???;
double b = ???;
double c = ???;
return (b*b > 4*a*c) ?
        2*a : 2*c;
```

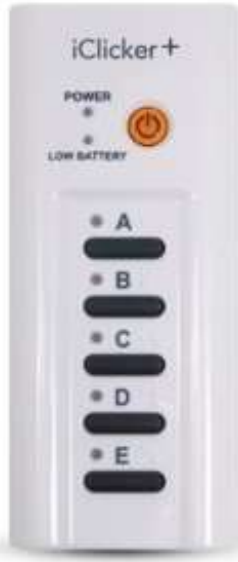
What is the static type of the expression **4*a**?

A. `int`

B. `double`

C. `boolean`

D. Indeterminate (depends on runtime values)



Warmup 2

```
int a = ???;
double b = ???;
double c = ???;
return (b*b > 4*a*c) ?
        2*a : 2*c;
```

What is the static type of the expression **4*a*c**?

A. `int`

B. `double`

C. `boolean`

D. Indeterminate (depends on runtime values)

Warmup 3

```
int a = ???;
double b = ???;
double c = ???;
return (b*b > 4*a*c) ?
        2*a : 2*c;
```

What is the static type of the expression **b*b > 4*a*c**?

A. `int`

B. `double`

C. `boolean`

D. Indeterminate (depends on runtime values)

Warmup 4

```
int a = ???;
double b = ???;
double c = ???;
return (b*b > 4*a*c) ?
        2*a : 2*c;
```

What is the static type of the **returned value**?

A. `int`

B. `double`

C. `boolean`

D. Indeterminate (depends on runtime values)

CS 2110
Lecture 2

Classes, objects

Reminders

A1 released

Submit discussion activity

Complete quiz 1 on Canvas

Complete syllabus quiz on Canvas

Roadmap

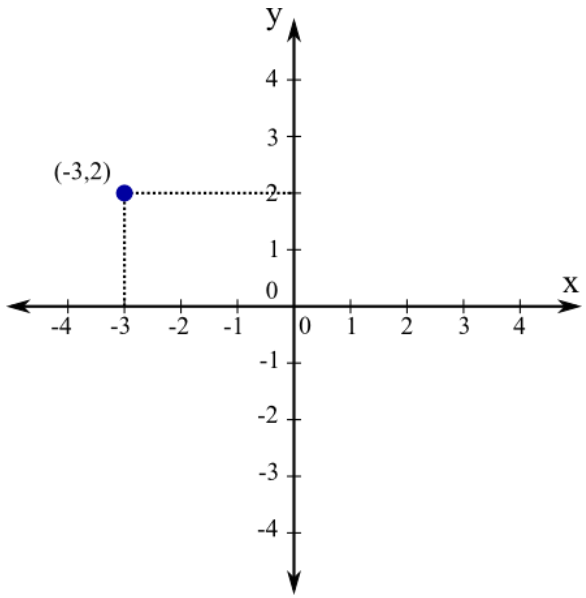
- Part I: OOP in Java
- Part II: Data structures
- Part III: Programming models

This lecture

- Classes
- Scope
- Reference types

Object-oriented programming in Java

User-defined types



- Java provides numbers and Booleans
- What type would you use for a 2D point?
  - Need to *aggregate* multiple primitive values
  - Need to define new operations

- Can make new types by defining **state** and **behavior**

- Vocab: **state** = "what distinguishes one value of a type from another"
  - Think of as the *data* stored in a value

Classes and objects

- A **class** defines a *type* by specifying state and behavior
- Values of the type are called **instances** of the class
- Instances of classes are called **objects**

- Analogy: *classes* are blueprints for objects.
- An *object* is like a house built according to those blueprints

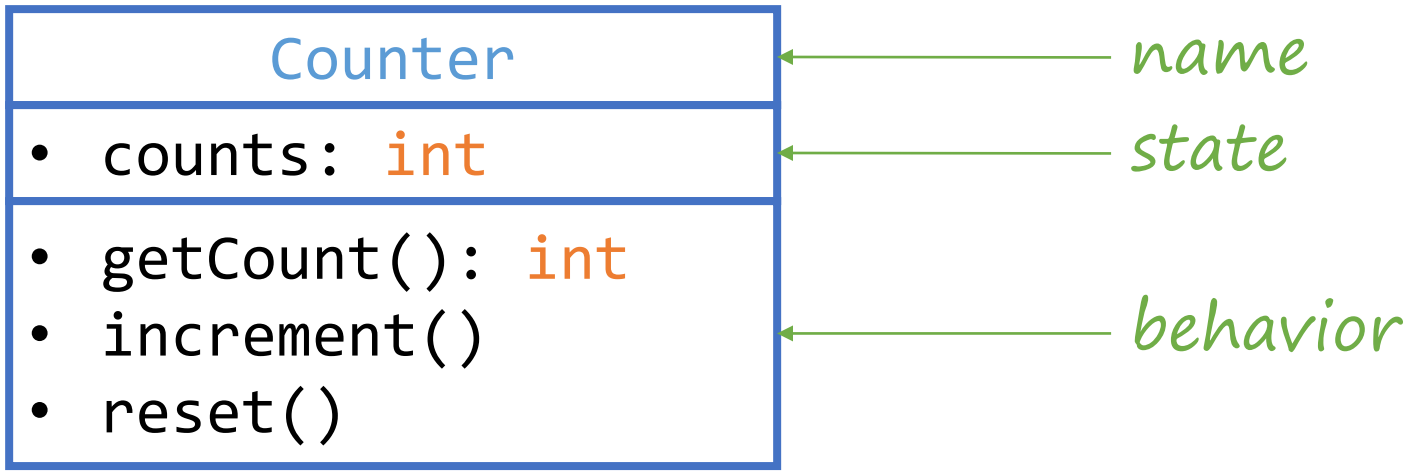
- Duality: classes are defined at compile-time, objects exist at runtime

Object-oriented modeling

- What **behaviors** does a counter exhibit?

- What **state** can a counter keep track of to provide those behaviors?

Class diagram (compile-time)



Class definition syntax

```
class Counter {
    int counts = 0;

    int getCount() {
        return this.counts;
    }
    void increment() {
        this.counts += 1;
    }
    void reset() {
        this.counts = 0;
    }
}
```

Let's unpack that…

- **Fields** are variables that will live inside of objects
  - Aka "attributes", "properties", "member variables"
  - Initialized when an object is created
- **Methods** are functions that can access an object's fields
  - Aka "member functions"
- `this` is a "magic variable" that refers to the *object* a method was invoked on
  - Can create many instances of a class, each with its own copy of state
  - "Which counter's count? **This** counter's count!"

Creating objects and invoking behavior

new-expression

```
Counter c;
c = new Counter();
```

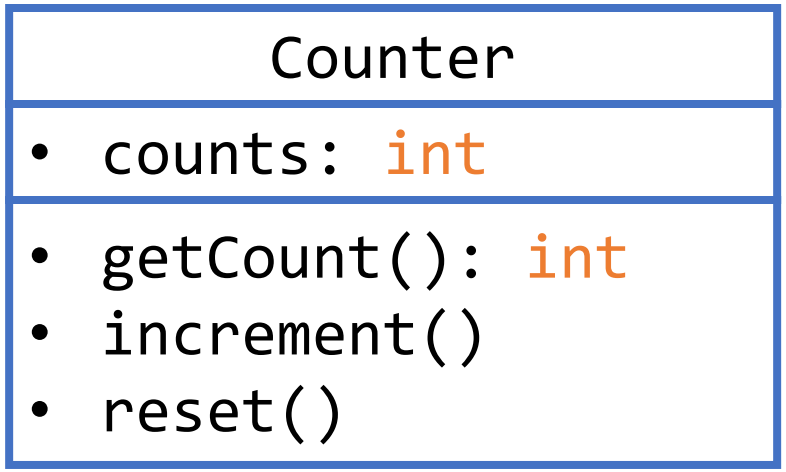
Method invocation

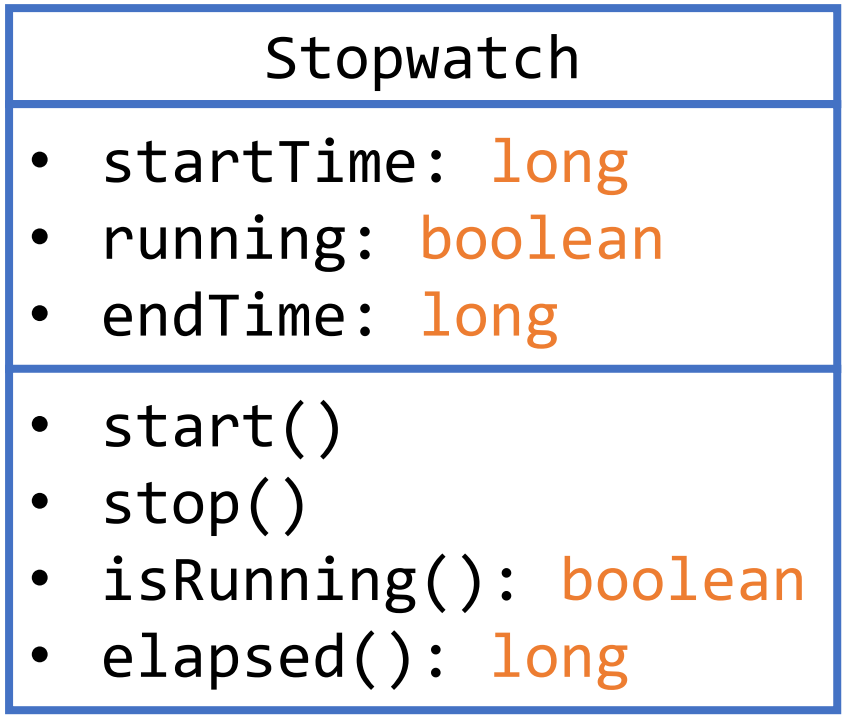
```
int n = c.getCount();

c.increment();
c.reset();
```

Exercise: Stopwatch class

- **Draw** a class diagram for a Stopwatch class
  - Behavior:
    1. Start counting
    2. Stop counting
    3. Get whether currently counting
    4. Get elapsed time in ns
- **Implement** using `System.nanoTime()`
  - Returns a `long` that counts ns
- Write client code to make and use a stopwatch



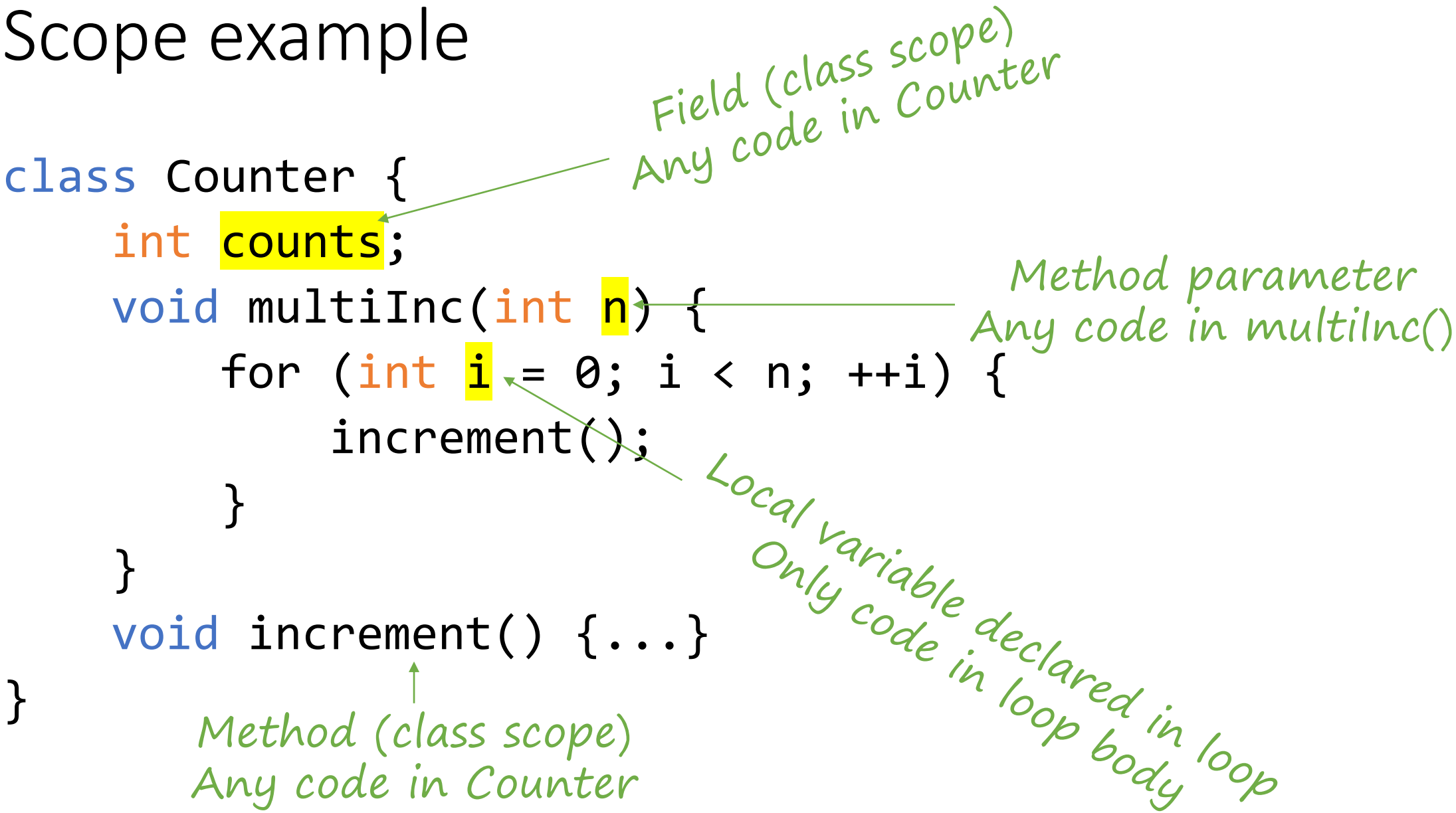

Stopwatch
• startTime: long
• running: boolean
• endTime: long
• start()
• stop()
• isRunning(): boolean
• elapsed(): long

Variable scope

- Compile-time: which code is allowed to access a variable?
- Runtime: when are variables created and destroyed?

- Every block { } creates a new scope
  - Class scope
  - Method scope
  - Else-branch scope
  - …
- Variables may be used in a statement if they were declared above in the same scope or an outer scope

Scope example



Java can infer `this` in methods

```
class Counter {
    int counts = 0;

    int getCount() {
        return this.counts;
    }
    void increment() {
        this.counts += 1;
    }
    void reset() {
        this.counts = 0;
    }
}
```

```
class Counter {
    int counts = 0;

    int getCount() {
        return counts;
    }
    void increment() {
        counts += 1;
    }
    void reset() {
        counts = 0;
    }
}
```

Poll


```
class MealCard {
  int balance;
  void pay(int price) {
    if (balance >= price) {
      price -= balance;
      boolean paid = true;
    }
    println(paid);
} }
```

Suppose `mc` is a `MealCard` with a balance of 5. What is printed by the call `mc.pay(10)`?

A. "true"
B. "false"
C. Runtime error: `paid` is not initialized
D. Compile-time error: `paid` is not in scope
E. Compile-time error: `balance` is not in scope

Shadowing

- Method parameters and local variables may have the same name as fields
- Local variable takes precedence *when it is in scope* (“inside-out rule”)
  - Field would then be “shadowed”
- To refer to shadowed field, use the object’s self reference, `this`

```
class Counter {
  int counts;
  void setCounts(
      int counts) {
    this.counts = counts;
  }
  int counts() {
    return counts;
  }
}
```

Reference semantics

Java values come in two flavors

Value semantics

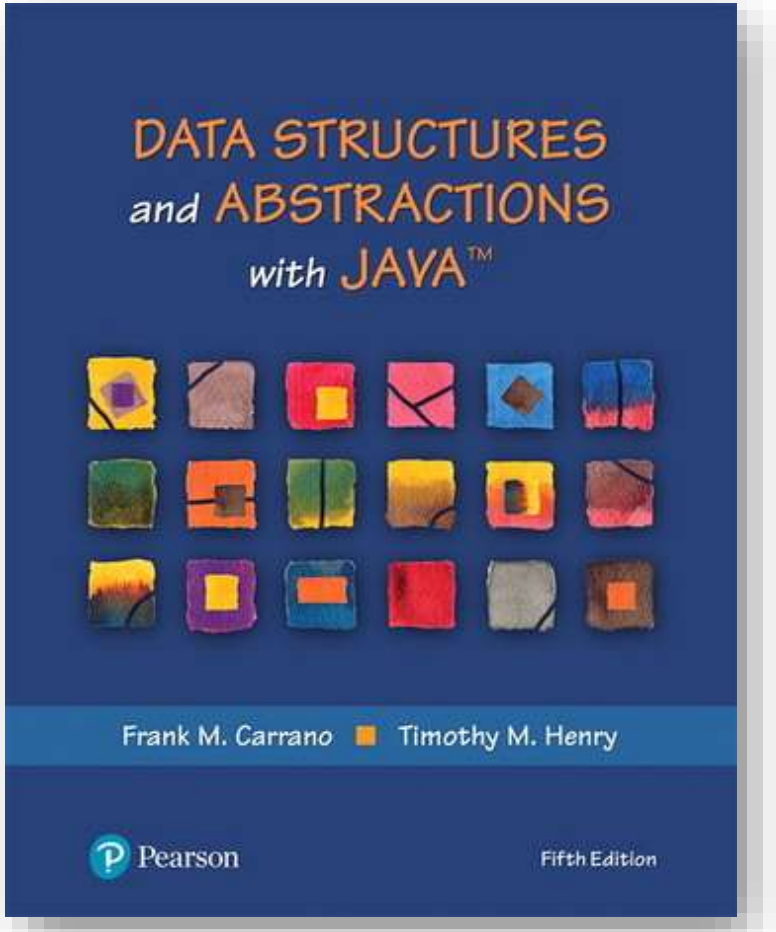
- Primitive types
- Values stored directly in variables
- Copies of the same value are indistinguishable

Reference semantics

- Classes and arrays
- Values (objects) exist on their own
  - Don’t “live” in variables
- Variables *point to* objects
- Different objects can be distinguished (even if state is equal)
- Possible for variable to reference nothing (`null`)

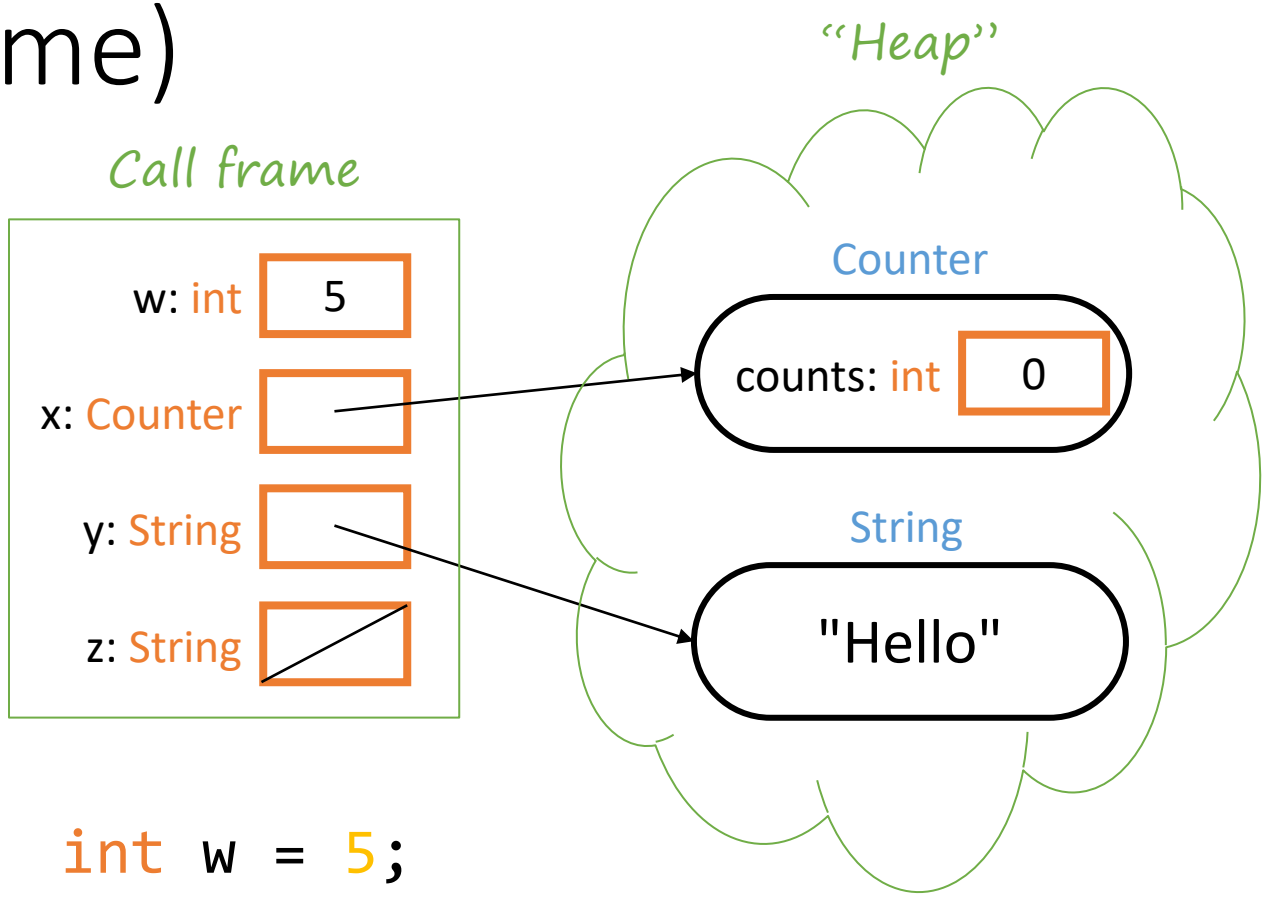
Analogy

- Sharing a paper textbook is not the same as having two separate copies



Object diagrams (runtime)

- Rules
  - Rectangular boxes: **variables**
  - Primitive values go directly in variable boxes
  - Rounded boxes: **objects**
  - Object boxes contain **fields** (or a string representation)
  - Variable boxes point to objects with arrows, or are crossed out if **null** (never empty)
  - Variables NEVER point to other variables




```
int w = 5;
Counter x = new Counter();
String y = "Hello";
String z = null;
```

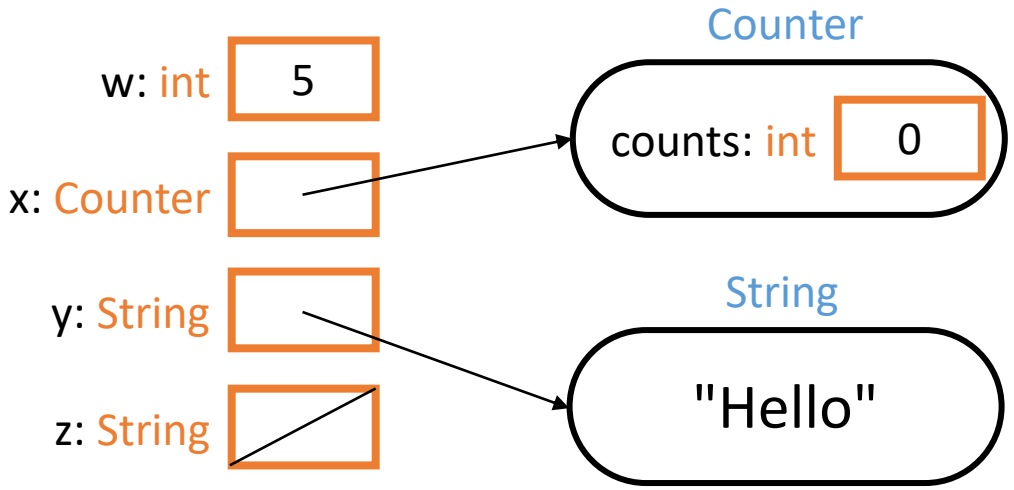
Assignment

- *<var> = <primitive literal>*
  - Write the literal in *var*'s box
- *<var2> = <var1>*
  - Copy the contents of *var1*'s box (primitive value or arrow) into *var2*'s box
- *<var> = <expr>*
  - If *expr*'s static type is primitive, write value in *var*'s box
  - If *expr*'s static type is reference, draw arrow to value (on heap) in *var*'s box

- `a = 5;`
- `b = a`
- `c = new Counter();`
  `d = c;`
- `e = "brother".substring(2);`

Practice: object diagrams

(for reference)



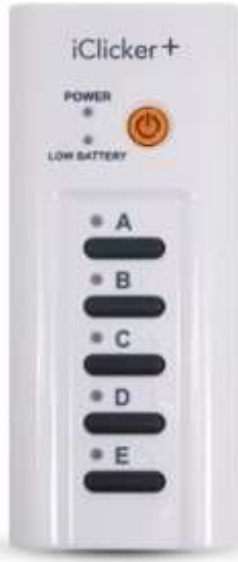
Draw the object diagram for the following code:

```
Counter c1 = new Counter();
Counter c2 = c1;
c1.increment();
c2 = new Counter();
```

Poll

What is `c1.count`, according to your diagram?

A. 0
B. 1
C. 2
D. null



```
Counter c1 = new Counter();
Counter c2 = c1;
c1.increment();
c2 = new Counter();
```