Lecture 8: Classes and Encapsulation

CS 2110, Matt Eichhorn and Leah Perlmutter

September 18, 2025

Announcements

- Equitable teaching and learning environment

Announcements

- A2 clarification (problem 2.3: Describe your tests)
- A4 released!
  - Ask for clarification as needed

Today's Learning Outcomes

Let’s jump into Object Oriented Programming!

1. Given the description of a class, identify its state and behaviors and use this to sketch its fields and public method signatures.
2. Identify the invariant of a class and write a method to assert that this invariant is satisfied.
3. Determine the scope and lifetime of a variable.
4. Explain the object-oriented principle of encapsulation and its benefits. Identify examples of proper and improper encapsulation in provided code.

Object-oriented programming in Java

Recall: Classes and Objects (lec02)

- class - code that specifies blueprint for non-primitive type
  - State (fields)
  - Behaviors (methods)

<---------- Template

defined at compile-time

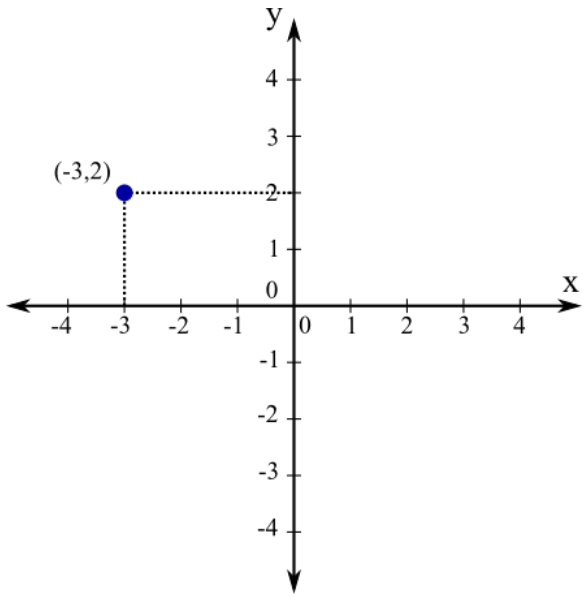
- object - an instance of a non-primitive type

<---------- Thing created using the template

exist at runtime

Today we'll write code to define classes!

User-defined types



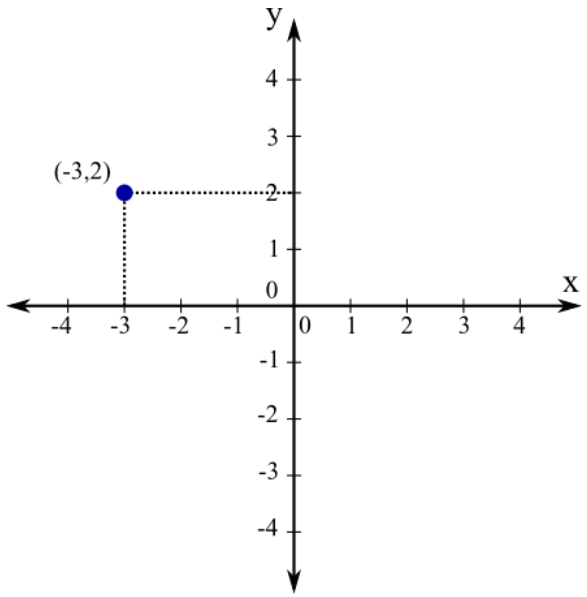
- Java provides numbers and Booleans
- What type would you use for a 2D point?
  - Need to *aggregate* multiple primitive values
  - Need to define new operations

- 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

User-defined types

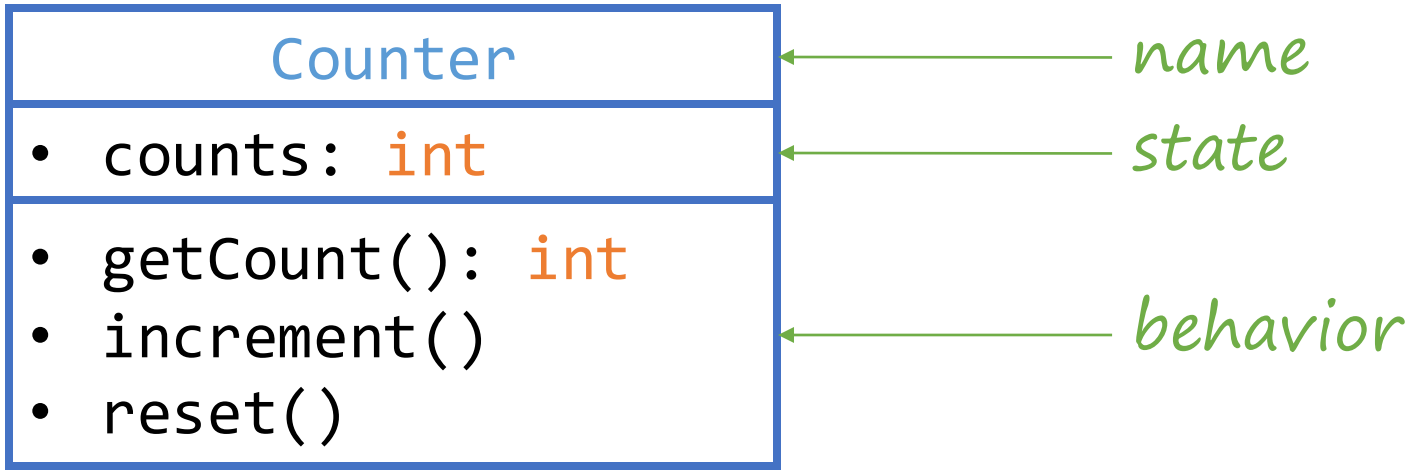
- Brainstorm:
  What **behavior** might a 2D point provide?
- …



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)



Demo: Defining a class

Class definition syntax

```
public class Counter {
    private int counts;

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

fields (state)

methods (behavior)

no more `static`

Not shown:
Javadoc comments

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 (aka its "target")
  - 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

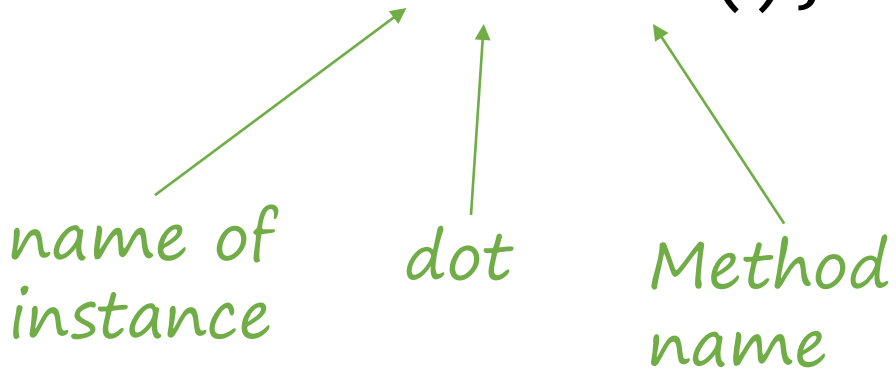
new-expression

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

Method invocation

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

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



Demo

Poll Everywhere

PollEv.com/leahp text **leahp** to **22333**

Poll: what value does n have?

```
Counter c = new Counter();
int n = c.getCount();
// n == 0
c.increment();
// What is n now?
```

A. 0

B. 1

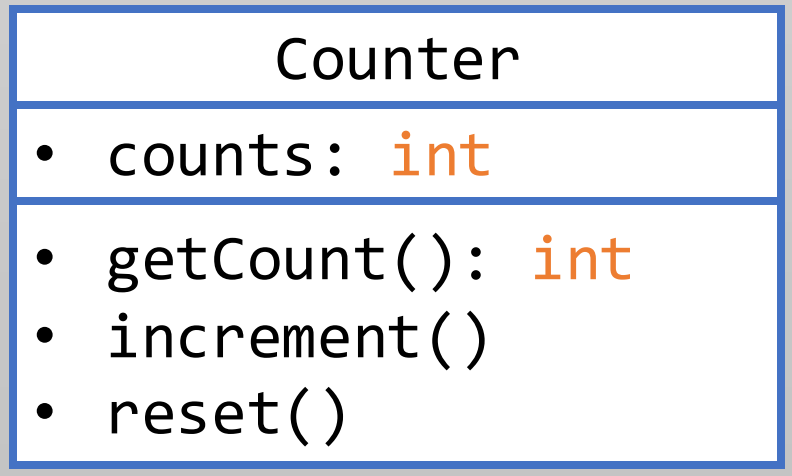
C. 2

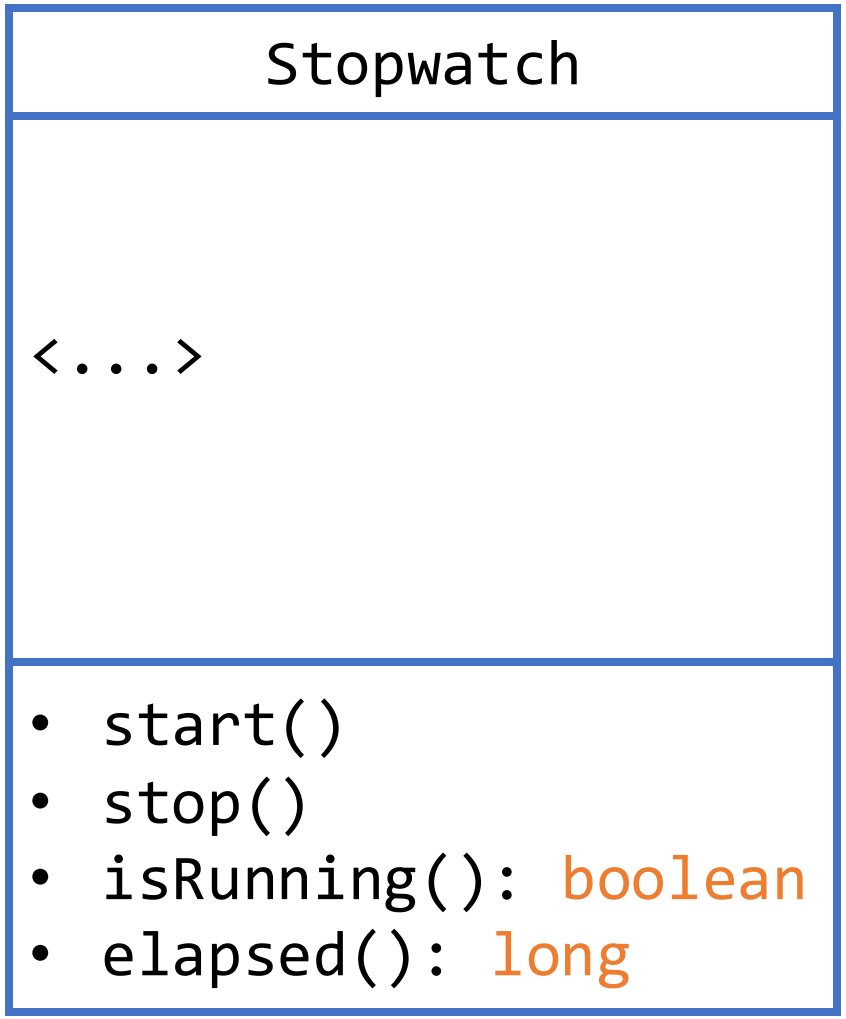
D. Something else

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

(Counter diagram, for reference)




Stopwatch
<...>
• start()
• stop()
• isRunning(): boolean
• elapsed(): long

Classes and Scope

Variable scope and classes

```
public class ScopeDemo {
  public static void foo(int x) {
    System.out.println(x);
  }
  public static void main(String[] args) {
    int x = 3;
    for (int i = 100; i < 110; i++) {
       foo(i);
    }
    System.out.println(i); //*
  }
}
```

- First, let's review scope…
- Relationship between x in main and x in foo?
- What happens in the line marked with *

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

```
public class Counter {
    private int counts;
    public void multiInc(int n) {
        for (int i = 0; i < n; ++i) {
            increment();
        }
    }
    public void increment() {...}
}
```

Field (class scope)
Any code in Counter

Method parameter
Any code in multiInc()

Local variable declared in loop
Only code in loop body

public method
Any code in Counter
Any code that imports Counter

Java can infer `this` in methods

```
public class Counter {
    private int counts = 0;

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

```
public class Counter {
    private int counts = 0;

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

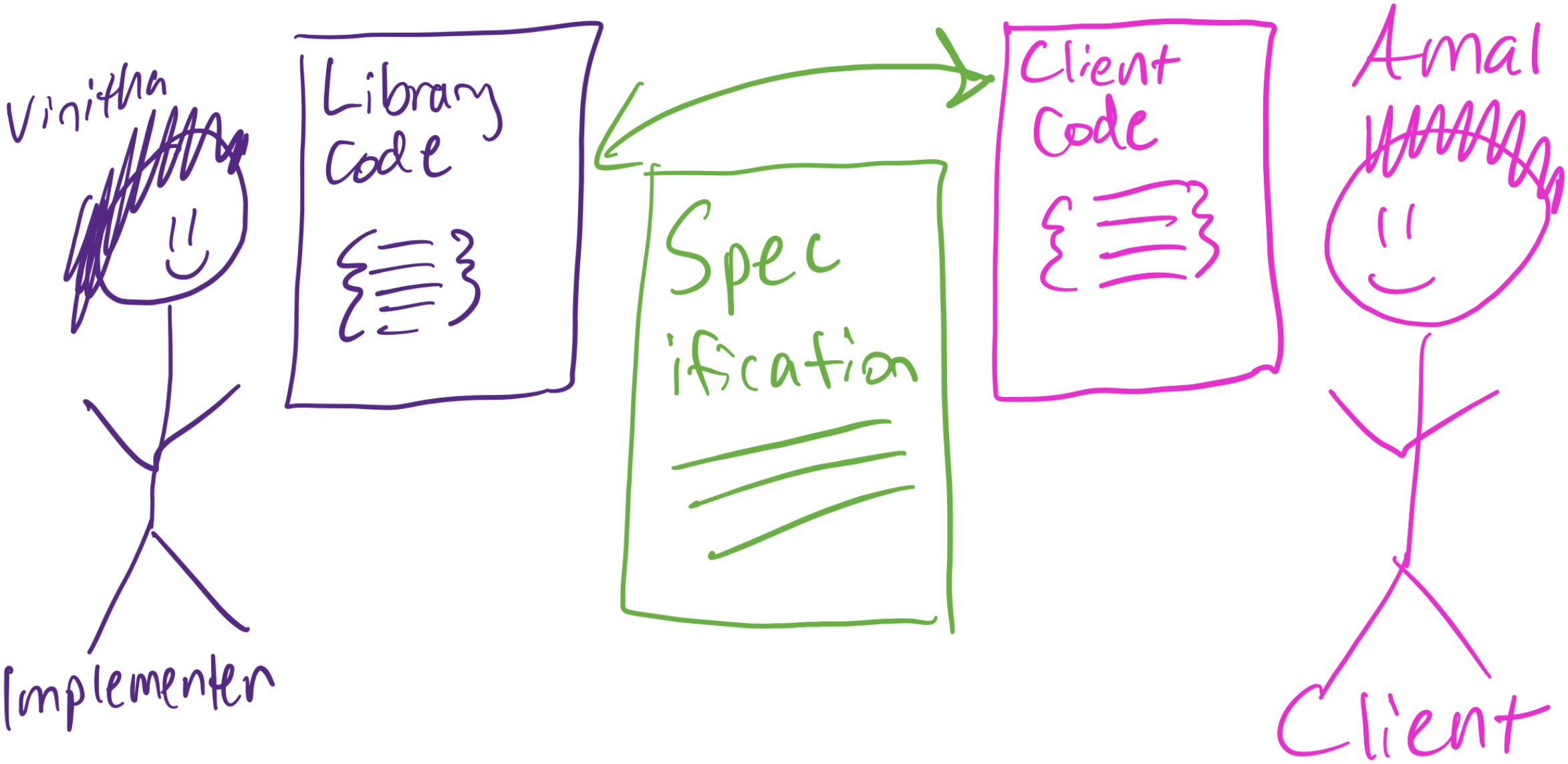
Shadowing

- 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`

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

Client and Implementer (Revisited)

Why Specifications? The Client and the Implementer



Client and implementer

- Refers to a **role** with respect to a class in a particular context
  - We are all *clients* of the String class
  - None of us is the *implementer* of the String class

  - I am the *implementer* of my Counter class when writing code in "Counter.java"
  - I am the *client* of my Counter class when writing code in the main method (or anywhere else)
- You will play both roles, sometimes for the same type
  - Practice "splitting your brain" to adopt the appropriate role

Responsibilities

- Implementer must maintain class invariant, provide correct behavior
- Client should be able to use class *for any purpose* and never get incorrect behavior

How can implementer prevent clients from breaking things?

Abstractions, Invariants, and constructors

Object state

- Recall from lecture 1: A **type** is concerned with…
  - What values are allowed?
  - What operations can be performed on those values?

- Is a Counter object with “`counts=10000`” valid?

- What about a dictionary that is not in alphabetical order?

The Counter: An Abstraction

- Abstraction of the counter – a mathematical data model that represents the features of the counter that we most care about
  - States
  - Behaviors
- An abstraction is NOT code
- Our code can be based on the abstraction

- Write down an abstraction of a counter…

The Counter: An Abstraction

- Count is an integer from 0 to 9999
- We can increment the count by 1
- We can reset the counter

Implementing the abstract counter as code

- Field `counts`
  - Some kind of integer:
    - `int`
    - `long`
    - `BigInteger`

- There are multiple concrete (code) implementations of the same abstract counter
- To write code, we have to make a design decision and pick one

Implementing the abstract counter as code

- Let's say we settled on int for the count

Counter
• counts: int
• getCount(): int • increment() • reset()

- Is this good enough? …

Consider the Counter example again

- State representation:
  `int counts;`
- Allowed states:
  [-2147483648, 2147483647]

- Which states *should* be allowed?
- Could implementations of behavior yield disallowed states?

Class invariants

- Invariant: a statement that should always be true
  - **Class invariant**: relationship between fields; truthfulness not affected by calling methods
    - Example: "`counts` is non-negative", aka `counts >= 0`
  - (**Loop invariant**: relationship between local variables; truthfulness not affected by loop iterations)
- Typically, invariants are expressed as comments; programmer is responsible for enforcing them
  - by **defensively** writing code to *check* invariants and catch bugs earlier
  - (Some languages can verify invariants; see Dafny, Java Markup Language)

Demo: Improving Counter

Types and invariants

- **Static typing** enforces common invariants automatically
  - "`int counts` *is an integer*"
  - "`String name` *is a string of characters*"

- If an invariant concerning a field is captured in the field's type, you *do not* need to document that invariant separately
  - Comments redundant with code are a maintenance burden
  - Tools for refactoring code do not always understand comments

Creating objects

- How to initialize fields to represent a state *specified by a user*?

- How to establish that the *class invariant is satisfied* from the start?

Constructor

- Syntax
  - Like a method, but no return value, and name matches name of class
  - Invoked with new-expression
  - Can delegate to other constructors by calling `this(…)`
- Job: truthify the class invariant
  - Initialize all field values
- Default constructor
  - No parameters
  - Initializes all fields to default values

```
public class Counter {
    private int counts;
    public Counter() {
        counts = 0;
    }
    // ...
}
```

Demo: Adding a Counter constructor

Discussing a Class Design Example: Fraction

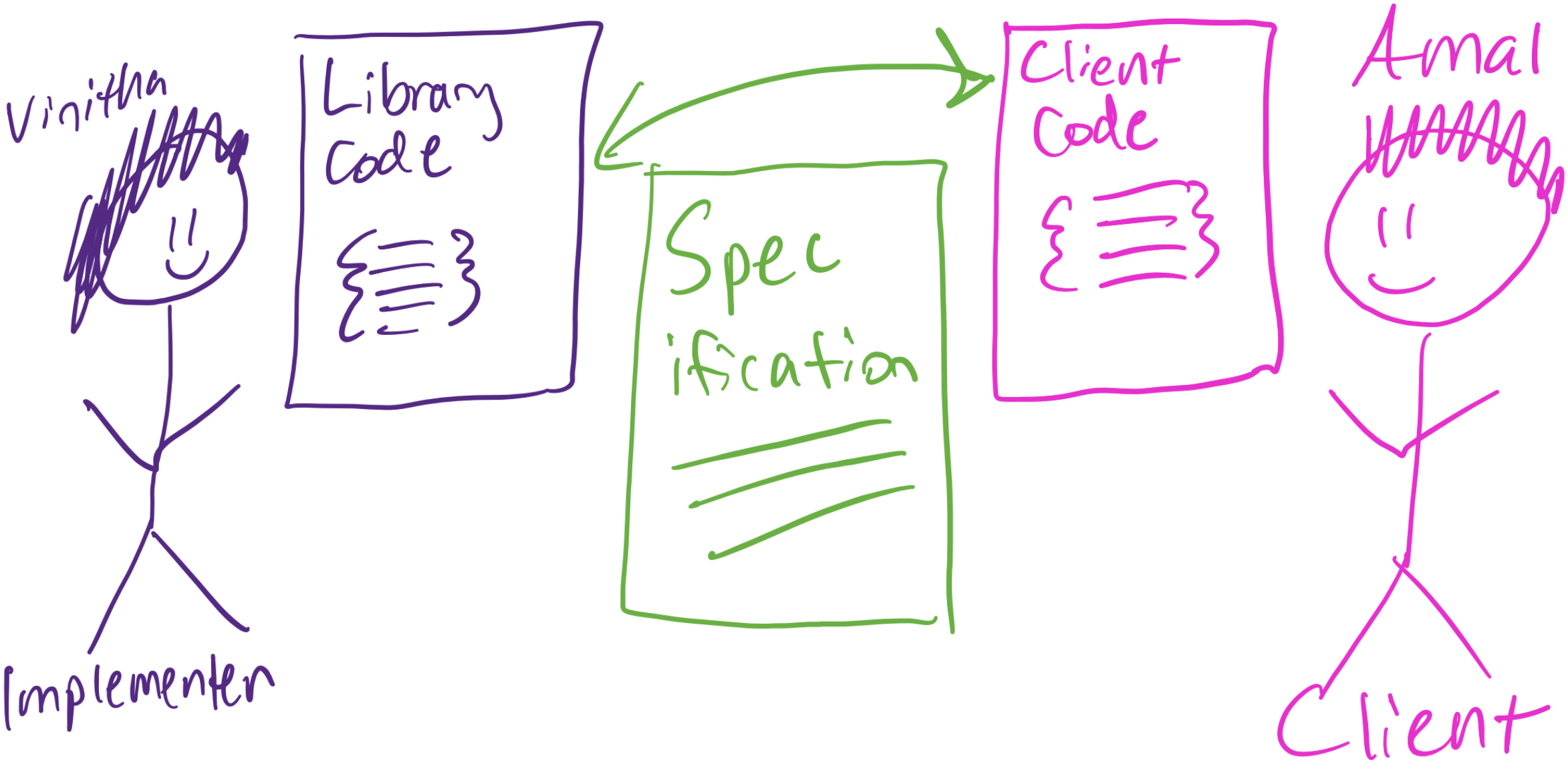
- What fields could represent a fraction?
  - …
- Are any field values invalid?
  - …

Question: should these be the same Fraction?

- It depends on what **`Fraction`** is designed to represent!
  - i.e. what is our abstraction of a fraction
- For middle school math, we want to be able to represent un-reduced fractions
- For rational numbers, we want no more than one representation for each point on the number line

Encapsulation

Why Specifications? The Client and the Implementer



Encapsulation

- Programming languages can help us protect a class's state
  - What if state were invisible to users? What if they could only invoke (a subset of) objects' behaviors?
  - Theme: giving up flexibility to achieve reliability
- **Access modifiers**
  - `public`: Anyone can access fields / invoke methods
  - `private`: Only the class implementation can access fields / invoke methods

Encapsulated Counter

```
public class Counter {
    /** Class invariant: `counts` is in [0,9999]. */
    private int counts;

    public Counter() { counts = 0; }
    public int getCount() { return counts; }
    public void reset() { counts = 0; }
    public void increment() {
        if (counts == 9999) { counts = 0; }
        else { counts += 1; }
    }
}
```

Access modifier recommendations

- If class is public, fields should always be private
- “Helper” methods should be private
- Public methods should provide meaningful behavior to clients
  - Can become widely used if code is open-sourced
  - Maintenance burden: cannot change behavior without breaking clients

Clients, implementers, and encapsulation

- Encapsulation allows implementer to hide implementation details from client so that the client can't accidentally break the class invariant
- Method header documentation (our beloved Javadoc comments) communicate the specification to the client

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow…

Announcements

- A4 released today, due Wednesday