Specifications, testing, abstraction
I can’t do this, there’s too much Jargon here
Pause. Breathe…
Terms So Far

- Java syntax specific
  - this
  - final
  - static
  - main
  - class
  - void
  - public/private

- Concepts
  - Value/Reference Semantics
  - Primitive/Class Types
  - Specifications/Invariants
  - Constructors/Getters/Setters
  - Scope
Some stuff I’ve missed in some part

- Arrays in Java
- Scope
  - Let’s just Demo this!
What is a bug?
How do we catch them?
Who gets the blame?
Fractions revisited

/** Represents a rational number. */
class Fraction {
    /** The numerator of the quotient representation `num/den`. */
    int num;

    /** The denominator. Must be positive, and the GCD with `num` must be 1. */
    int den;

    ...
}

• Is this Fraction object a bug?

```java
Fraction num: 1
int den: 0
```
Demo

Fraction f =
    new Fraction(0, 1);

f.den = 0;

println(f.toDouble());
Whose fault is this?

The person who wrote `Fraction.java`

The person who wrote `f.den = 0`
Fun fact

Opening IntelliJ involves interactions between objects from…

• 600 different classes?
• 6,000 different classes?
• 60,000 different classes!

You can find the IntelliJ source code here: https://github.com/JetBrains/intelliJ-community/tree/89e4cdaeb1522688405dcd749de5f469ccaa009a
Dividing responsibilities

**Client**
- Uses classes to solve problems
  - Constructs objects
  - Invokes methods on objects
  - Could be writing code anywhere
- Could be anyone

**Implementer**
- Implements a class’s behavior in terms of its state
  - Writes code in class’s “.java” file
  - Might be a *client* of other classes to get the job done
- An assigned duty
Client vs. 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 Fraction class when writing code in “Fraction.java”
  - I am the *client* of my Fraction class when writing code in “FractionDemo.java” (or anywhere else)

- You will serve both roles, sometimes for the same type
  - Practice “splitting your brain” to adopt the appropriate role
Example Application: A calculator Application

Can we add Fraction support to this calculator app given that its Written in Java?
Whose the client, and whose the implementor in this case?

Classes are always simply building blocks of Java applications!
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?
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
Types of methods in here

- Constructors: Class methods that are initially allowed to construct the initial state of an object

- Getter methods: Class methods that can return/survey the state of an object but not allowed to modify it

- Setter methods: Class methods that can modify state
Encapsulated Counter

```java
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; }
    }
}
```
Accessibility recommendations

• If class is public, fields should always be private
• Public methods provide meaningful behavior to clients
  • Maintenance burden: cannot change behavior without breaking clients
• “Helper” methods should be private
What about implementer bugs?

- Want to know if invariant is ever violated
- Add a method to check whether the invariant is true
  - `private boolean checkInv() { ... }`
- **Assert** that the invariant is true
  - Want program to crash if ever false
  - Assert at end of every method
Demo: Fraction
Java’s `assert` statement

- `assert` `checkInv();`
- Crashes program if condition is false
- Not checked by default outside of unit tests!
  - Must add “-ea” to VM options in IntelliJ “Run configuration”
What is a bug?

• Our class invariant is never violated; are we bug-free?

• How can we verify that our methods behave correctly?
  • Given a method’s output, how do you decide whether it is right or wrong?
Specifications
Abstraction: what vs. how

• A *specification* says **what** a class or field represents or **what** a method should accomplish

• An *implementation* dictates **how** some behavior is achieved
  • More than one way to do it!

• It is *impossible* to verify an implementation without a spec
  • “Not even wrong”

• Not “just documentation” – specs define what it means to be a bug
Types of methods in here

- Constructors: Class methods that are initially allowed to construct the initial state of an object

- Getter methods: Class methods that can return/survey the state of an object but not allowed to modify it

- Setter methods: Class methods that can modify state
Verification

Testing
• Confirm that impl satisfies spec at particular input values
• Spec is used to determine expected outputs

Code review
• Human confirms whether impl logic appears to satisfy spec
• Without spec, can only look for “bad practices”; cannot say whether impl is right or wrong
  • Applies to TAs/Consultants!
Good specs: methods

• **Returns**: what is special about the return value?
  • **Creates**: the return value is a new object

• **Modifies/effects**: how does the method mutate its target or arguments?

• **Requires**: what is assumed about its target or arguments?

• Interpret *every* parameter

```java
/**
 * Return the area of a regular polygon with
 * `nSides` sides of length `sideLength`
 * Requires `nSides` is at least 3, `sideLength`
 * is non-negative.
 */

static double polygonArea(int nSides, double sideLength)
```
More examples of specs
Preconditions and postconditions

**Preconditions**
- Assumed to be true at start of method
  - *Undefined behavior* if violated
- Responsibility of the client
  - Implementer *may* assert to be defensive
- “Requires” clause
- Implicit
  - Class invariant is true
  - Arguments are non-null

**Postconditions**
- Promised to be true at end of method
- Responsibility of implementer
- “Returns” and “Modifies/Effects” clauses
- Implicit
  - Class invariant is true
Should you write a test case to check method behavior when preconditions are violated?

Yes

No
Good specs: fields

- Fields are private, so their specs are for the implementer
- Explain how fields represent the logical state
- Capture invariants

```cpp
/** Represents a rational number. */

class Fraction {

  /** The numerator of the quotient representation `num/den`. */
  int num;

  /** The denominator. Must be positive, and the GCD with `num` must be 1. */
  int den;

  ...
}
```
Specifications are a contract

Between client and implementer (public)
- If preconditions are violated, client is at fault
- Otherwise, class is buggy
- Cannot be changed without affecting every client

Between implementer and future maintainer (private)
- Guardrails for changing state representation or method implementations
- Violations are a bug even if all observable behavior (currently) works
When to write specs?

• Before writing any code!
  • How can you write code if you haven’t said what it’s supposed to do?

• Don’t think of as “documentation”
  • Specs define what behavior is “right”
  • Without specs, “all bugs are features”

• From here on, **all** methods, fields, and classes require a spec
Testing
Scopes of testing

**Unit testing**
Isolated modules

**Integration testing**
Several interacting modules

**End-to-end testing**
Full application
Why not just end-to-end testing?

• Could just verify that application meets user requirements
  • Bug = requirement not met
  • No need for intermediate notions of correctness, specs

• How would you debug? When would you debug?
  • Unit testing finds bugs sooner and isolates them
  • Incremental testing saves time!
Black box testing

• Verify method postconditions against their spec
  • Don’t even look at method implementation

• Try to pick inputs that explore “corner cases”
  • Remember: goal is to break your code, not baby it
JUnit

• JUnit assertions != Java assert statements
  • assertEquals()
  • assertTrue() / assertFalse()

• Argument order: expected, then actual

• Floating-point is tricky (see comment in A1Test)
Coverage

• Testing is not sufficient to *prove* correctness
• How confident are you that there aren’t lingering bugs that weren’t triggered by your test cases?

• Quantifying coverage for black-box testing is difficult
Glass box testing

• Look at implementation; choose inputs to trigger different branches
• Can measure “line coverage”
• Any code not covered is code where your *customer* will be the first person to ever run it, with no evidence that it’s expected to work
• Disadvantage: breaks abstraction barrier
  • Focus on breaking the how, not stressing the what
What is the minimum number of test cases needed for complete line coverage of Counter.increment()? 

0
1
2
9,999
10,000
Time to breathe

- Java syntax specific
  - this
  - final
  - static
  - main
  - class
  - void
  - public/private
  - arrays

- Concepts
  - Value/Reference Semantics
  - Primitive/Class Types
  - Classes/Objects/fields
  - Specifications/Invariants
  - Constructors/Getters/Setters
  - Scope
  - Junit/Black Box, Glass Box Testing/