Warmup: any concerns?

```java
/** Represents a finite, closed range
  * of real numbers. */
public class Interval {
  /** The smallest number in the
    * range; finite. */
  private final double lo;

  /** The largest number in the
    * range; finite and no less
    * than `lo`. */
  private final double hi;

  public Interval(double lo,
                  double hi) {
    this.hi = hi;
    this.lo = lo;
}

  public Interval intersect(
      Interval other) {
    return new Interval(
        Math.max(lo, other.lo),
        Math.min(hi, other.hi));
  }
}
```

Warmup

A. There is a problem related to **visibility** (access modifiers)

B. There is a problem related to **scope** or **shadowing**

C. There is a problem related to **mutability**

D. There is a problem related to preserving the **invariant**

E. There is a problem with **static types**

```java
/** Represents a finite, closed range
  * of real numbers. */
public class Interval {

  /** The smallest number in the range;
    * finite. */
  private final double lo;

  /** The largest number in the range;
    * finite and no less than `lo`. */
  private final double hi;

  public Interval(double lo, double hi) {
    this.hi = hi;
    this.lo = lo;
  }

  public Interval intersect(
      Interval other) {
    return new Interval(
        Math.max(lo, other.lo),
        Math.min(hi, other.hi));
  }
}
```

CS 2110 Lecture 4

Specifications, testing, abstraction

Reminders

A1 (late)

Quiz 2

Bugs

What is a bug?
How do we catch them?
Who gets the blame?

Defensive programming

Last time: Encapsulation

- Access modifiers protect an object's state against arbitrary *client* modification
- But what about buggy modification by implementers?

What about implementer bugs?

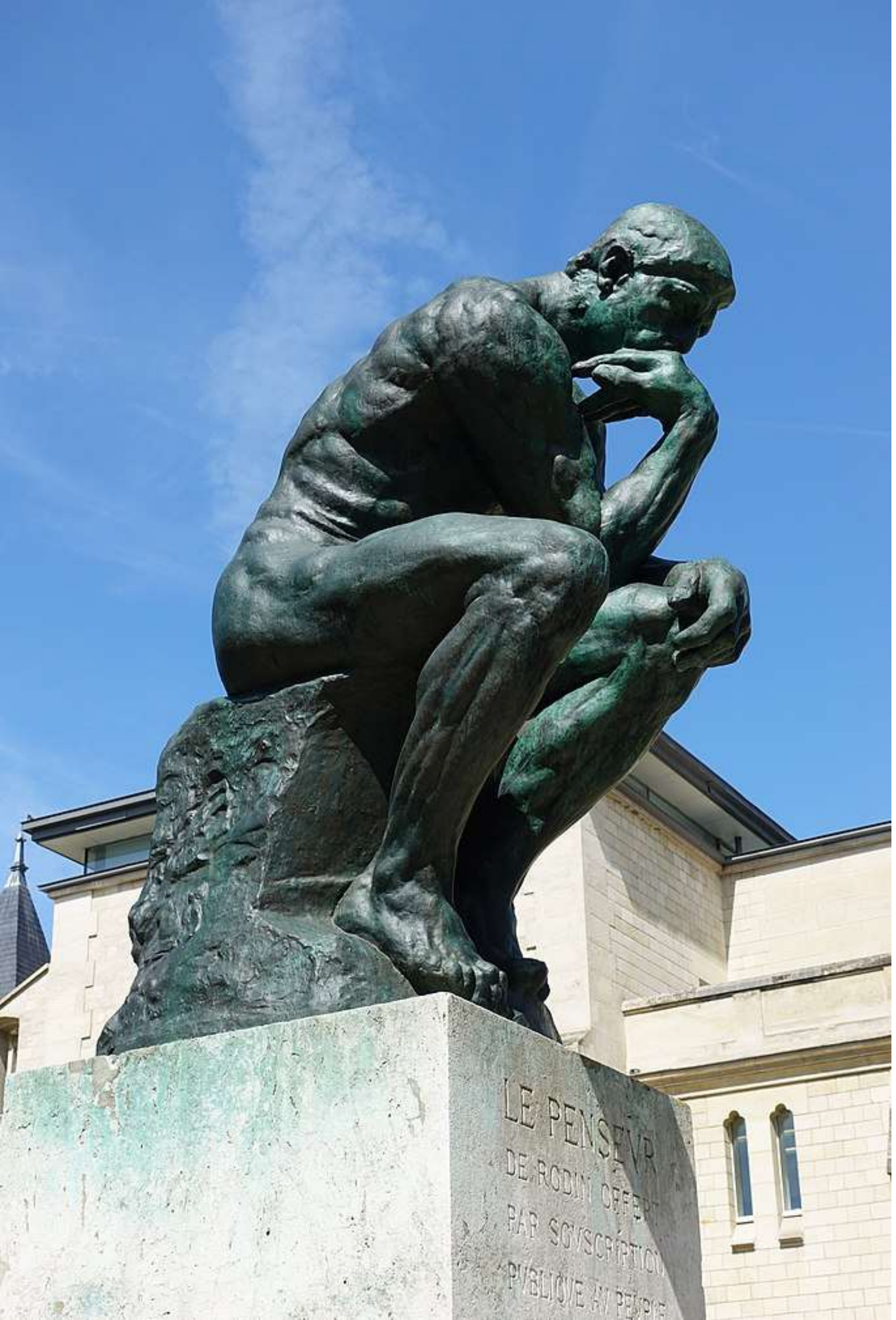
- Want to know if invariant is ever violated
- Add a method to check whether the invariant is true
  - `private void assertInv() { … }`
- **Assert** that the invariant is true
  - Want program to *crash* if ever false
  - Assert at end of every method

Java’s `assert` statement

- `assert` *<boolean expression>*`;`
- Crashes program if condition is false
- Not checked by default outside of unit tests!
  - Must add “-ea” to VM options in IntelliJ “Run configuration”

Demo: Fraction

- Define `checkInv()` using asserts
- Check invariant at end of constructor
- Check invariant at end of mutating methods



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

Example: JavaDoc

https://docs.oracle.com/en/java/javase/21/docs/api/index.html

Verification

Testing

- Confirm that impl satisfies spec at particular input values
- Spec is used to determine expected outputs (or properties thereof)

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

```
/**
 * 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

1. `A1.med3()`
2. `A1.intervalsOverlap()`
3. `Counter` (lec04 edition)
4. `Fraction` (lec04 edition)

A good spec doesn’t tell you how a method does its job;
it tells you how you could *verify* that it did its job right.

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

Demo: Asserting preconditions

- `A1.polygonArea()`

Poll

Should you write a test case to verify method behavior when preconditions are violated?

A. Yes
B. No

Good specs: fields

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

```
/** 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?

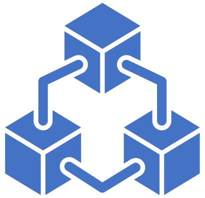
- Before writing any code!
  - How can you write code if you haven’t said what it’s supposed to do?
- Don’t think of as “documentation”
  - Specs define what behavior is “right”
  - Without specs, “all bugs are features”
- From here on, **all** methods, fields, and classes require a spec

Testing

Scopes of testing

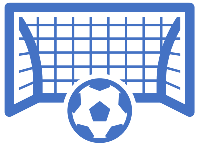
Unit testing

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

1. Read the spec
2. Think of a scenario (satisfying the preconditions)
   - Create supporting objects; call the method with arguments
3. Deduce properties of the output/effects implied by the postcondition
4. Assert that these properties are true
   - Write JUnit assertions

Tip: Given, when, then

- "Given *X*, when *Y* is done, then *Z* should happen."
  - Setup, operate, check ("arrange, act, assert")
  - Document the test's purpose in English (for JUnit, use `@DisplayName("…")`)

Examples:

- Given a Point in quadrant I, when it is rotated 90 deg CCW about the origin, then it should be in quadrant II.
- Given a Calendar representing Jan 31 in a leap year, when it is "rolled" by 1 month, then it should represent Feb 29.

Longer [example]

Scenario: User requests a sell before close of trading

- Given I have 100 shares of MSFT stock
  - And I have 150 shares of APPL stock
  - And the time is before close of trading

Setup

- When I ask to sell 20 shares of MSFT stock

Call the method-under-test

- Then I should have 80 shares of MSFT stock
  - And I should have 150 shares of APPL stock
  - And a sell order for 20 shares of MSFT stock should have been executed

3 asserts

Which scenarios?

- At least one “nominal case” with an obvious answer
- All off-nominal behavior (still satisfies preconditions, but spec prescribes special treatment)
- Several “edge cases”
  - Remember: goal is to break your code, not baby it

Common “edge cases”

Numbers

- 0
- Positive and negative
- Less than and greater than 1
- Min and max values
  - e.g. dates, times
- (Special floating-point values)

Collections

- Empty
- Single element
- Two elements
- 3+ elements (general case)
- Duplicates
- Sorted/unsorted
- Full

JUnit

- JUnit assertions != Java `assert` statements
  - `assertEquals()`
  - `assertTrue()` / `assertFalse()`
- Argument order: *expected*, then *actual*

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

Practice: Black-box tests

```
/** Return a solution `x` to the equation
  * `a*x^2 + b*x + c == 0`.  Requires `a != 0`,
  * `b^2 >= 4*a*c`. */
static double qSolve(double a, double b, double c) {
    …
}
```

Use Junit’s `assertEquals(expected, actual)`.

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

Poll

What is the minimum number of test cases needed for complete **line coverage** of `Counter.increment()`?

A. 0
B. 1
C. 2
D. 9,999
E. 10,000

```
void increment() {
    if (counts == 9999) {
        counts = 0;
    }
    else {
        counts += 1;
    }
}
```