Encapsulation

I. Encapsulation intro
   a. "Anti-feature": restricts you from doing things you could do otherwise
   b. Valuable for maintainability and correctness
II. Code size
   a. Intro programming class: few hundred lines per project (~2000 total)
   b. Simulating black holes and neutron stars at Cornell: few hundred thousand lines
   c. Web browsers, productivity software, operating systems: tens of millions of lines
III. Developing large systems
   a. Not written by one person, not written all at once
   b. Requires big teams
   c. Leverage libraries written by unrelated people
   d. Object-oriented languages promote maintainability, reusability
IV. Maintainability
   a. Doesn't unlock new algorithms; doesn't make code run faster
   b. Bigger codebases -> bigger problems
      i. Each programmer understands less of the total code
   c. Big teams require strong walls – defend portions of system against mistakes by programmers who don't (and can't) understand everything
V. Encapsulation
   a. Enclose something as if inside a capsule; barrier between inside and outside
   b. When programming, want to encapsulate implementation details
      i. Users need to know _____ something does, but not _____ it does it
   c. Transparency: good in government and finance (hard to hide corruption, fraud), bad in software (hard to change things without affecting others)
   d. Only reveal _____, not state
      i. And only ones you are willing to support long-term
      ii. OOP emphasizes verbs, not nouns
   e. Protect state from surprise modifications
      i. Ensure assumptions aren't violated
   f. Limitations for users = freedom for implementers
      i. Can change fields (optimization, new features) while still providing old behavior – users aren't broken by upgrades
VI. Demo: Fraction

```
class Fraction1 {
    int num;
    int den;
    double toDouble() {
        return (double) num / den;
    }
    void print() {
        System.out.println(num + "/" + den);
    }
```

```
}
```

a. Cast required for division
b. JShell: /open command
c. Zero denominator: not a valid rational number
    i. But field type is int, and 0 is a valid value of type int; how to prohibit it?

```
class Fraction2 {
    private int num;
    /** May not be 0. */
    private int den;
    public double toDouble() {
        return (double) num / den;
    }
    public void print() {
        System.out.println(num + "/" + den);
    }
    /** Precondition: d may not be 0. */
    public Fraction2(int n, int d) {
        assert d != 0;
        num= n;
        den= d;
    }
    public Fraction2(int n) {
        this(n, 1);
    }
}
```

d. Fields can be marked private
    i. Can't read fields
    ii. Can't reassign fields – protected against irrational state
e. Comment documents restrictions on values
f. Constructors (marked public)
    i. Can initialize objects when created
g. Enable assertions in JShell: "-R -ea" flags

VII. Access modifiers
a. private: can only be accessed by code in same _____ (constructors and methods)
    i. Free to change (changes confined to one file)
b. public: any code can read or write field, given reference to an object
    i. Must commit to maintaining (cannot change)
c. Default: accessible to other code in same _____
    i. Must coordinate with package maintainer before changing
d. protected: [deferred to future lecture]
    i. Must commit to maintaining (cannot change)
e. Can be applied to fields, methods, constructors, entire classes

VIII. Constructors
a. Invoked when creating a new object
b. Purpose: initialize an object's state

        i. Default field values aren't good enough – may violate requirements/assumptions needed for methods to provide sensible behavior
    c. Consolidates responsibility – only class maintainer needs to understand restrictions and relationships among fields, not every user

IX. Constructor syntax
    a. Resembles a method declaration
    b. Name must match class name (starts with capital | lower-case letter)
    c. No return type (not even void)
    d. Can only be invoked as part of a "new expression"
        i. Allocates space for new object and establishes its class
        ii. Constructor is executed on new object
        iii. _____ of object is returned as value of expression
    e. If constructor fails, address is never returned
        i. Impossible to interact with objects that were not successfully constructed

X. Default constructors
    a. If no constructors are defined in code, Java automatically inserts a default constructor
        i. Takes no arguments
        ii. Leaves fields at default values
    b. If any constructors are explicitly defined, then default constructor will *not* be generated
        i. Users *must* use one of the defined ones

XI. Overloading constructors
    a. May want to give users multiple ways to specify properties of a new object
    b. Distinguished by signature (types of parameters)
    c. May defer to one another (aka "chaining")
        i. Invoke using "this" keyword instead of class name (one of many uses of "this")
        ii. If done, must happen on _____ of constructor
        iii. Common pattern: one "full" or "raw" constructor whose parameters closely match fields (may be private). Additional constructors provide convenience by assuming default values or supporting other argument types.
    d. Requirement of all constructors: initialize fields so object is ready and safe for use.

XII. Getters and setters
    a. If all fields are private (and initialized in constructor), class is safe, but not useful
    b. Implement behavior (via methods) so users can benefit from data stored in fields
    c. "getter" method: allow users to read field value
    d. "setter" method: allow users to write field value
    e. Getters and setters are | are not in the spirit of encapsulation
        i. If they exist for each field, not much information is being hidden
    f. Advantages vs. public fields:
        i. Can separate read access from write access
        ii. Can accommodate some changes to field definition
        iii. Provides a "hook" for running additional code when property is accessed
            1. Audit uses of property
            2. Validate arguments in setters

3. Trigger object housekeeping tasks

g. Disadvantages of "traditional" getters and setters
   i. "get" and "set" names suggest field access, rather than abstract behavior
   ii. Names of setters may not reflect all changes that occur (multiple fields may be changed)

h. Traditional getters and setters are very common in standard Java libraries, but now considered an "anti-pattern" in most contexts

i. Alternative advice:
   i. Choose names that don't imply field access (drop "get" prefix from getters)
      1. Methods to query properties of objects (aka "getters") are otherwise fine!
   ii. Don't add behavior that isn't necessary and that you're not willing to maintain
      1. Focus on behavioral modeling, not field manipulation
   iii. Immutability (no setters) is desirable – see CS 3110

XIII. Demo: Fraction3

```
public class Fraction3 {
    private int num;
    /** The denominator. May not be 0.
     * The GCD of num and den must be 1, i.e. the fraction must be in
reduced form. */
    private int den;

    /** Construct a fraction with numerator n and denominator d.
     * Precondition: d may not be 0. */
    public Fraction3(int n, int d) {
        assert d != 0;
        num= n;
        den= d;
        reduce();
    }
    /** Construct a fraction with numerator n and denominator 1. */
    public Fraction3(int n) {
        this(n, 1);
    }
    /** Make this fraction be in reduced form. */
    private void reduce() {
        int g= gcd(num, den);
        num= num / g;
        den= den / g;
    }
    /** Return a representation of this fraction as a double. */
    public double toDouble() {
        return (double) num / den;
    }
    /** Print this fraction to standard out. */
    public void print() {
        System.out.println(num + "/" + den);
```

```
    }
    /** Add f into this fraction. */
    public void add(Fraction3 f) {
        // n1/d1 + n2/d2 = (n1*d2 + n2*d1) / (d1*d2)
        num= num * f.den + f.num * den;
        den= den * f.den;
        reduce();
    }
    /** Return the GCD of a and b. */
    private int gcd(int a, int b) { … }
}
```

a. Guarantees reduced form
    i. Relationship between two fields
    ii. Users cannot be trusted to maintain this property through public field access
    iii. Class is responsible for maintaining this by calling a "reduce" method when state changes, before releasing control back to user
b. Reduction occurs in constructor (must start with property true) and in mutating "add" method
c. A setNum() method would be bad – object might not be in reduced form (and if the setter called "reduce", the numerator wouldn't match what it was set to)
    i. Don't poke holes in abstraction

XIV. JavaHyperText reading

a. Access modifier (public, private), package
b. Constructor (call, default), new-expression
c. Overload
d. Getter setter