CS 2110
Lecture 3
Objects cont., encapsulation
Reminders

A1 due tomorrow
A2 released tomorrow
Somethings before we Start

- Notes about testing in Java and opening projects in IntelliJ
- There will be some additional OH tomorrow (before A1 is due)
- Am I missing something here?
Part 1 of this lecture

- Reference semantics
- Invariants & constructors
- Access modifiers
Roadmap

- Reference Semantics, Encapsulation • Lecture 3
- Specifications And Testing • Lecture 4
- Polymorphism and Inheritance • Lecture 5 and 6
Review: Primitive types vs. classes

Primitive types
• A single value (number or Boolean)
• Compute using **operators**

Classes
• *State* (set of **fields**) can aggregate multiple values
• Compute by invoking **methods** (whose implementations can operate on state)
Class diagram for Counter (compile-time)

Counter
- counts: int
- getCount(): int
- increment()
- reset()
Java values come in two flavors

<table>
<thead>
<tr>
<th>Value semantics</th>
<th>Reference semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Primitive types</td>
<td>• Classes and arrays</td>
</tr>
<tr>
<td>• Values stored directly in variables</td>
<td>• Values (objects) exist on their own</td>
</tr>
<tr>
<td>• Copies of the same value are</td>
<td>• Don’t “live” in variables</td>
</tr>
<tr>
<td>indistinguishable</td>
<td>• Variables <em>point to</em> objects</td>
</tr>
<tr>
<td></td>
<td>• Different objects can be</td>
</tr>
<tr>
<td></td>
<td>distinguished (even if state is</td>
</tr>
<tr>
<td></td>
<td>equal)</td>
</tr>
<tr>
<td></td>
<td>• Possible for variable to reference</td>
</tr>
<tr>
<td></td>
<td>nothing (<em>null</em>)</td>
</tr>
</tbody>
</table>
Object diagrams (runtime) [Our Format Requirements]

• 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
  • Variables NEVER point to other variables

```java
int w = 5;
Counter x = new Counter();
String y = "Hello";
String z = null;
```
Java Visualizer Plugin for IntelliJ

There’s a decent plugin for IntelliJ that helps you trace the state of your program. (I’m using a customized version of it in my demos)

- You can find it here if you want it: https://plugins.jetbrains.com/plugin/11512-java-visualizer

- Note that it has some limitations (example, it represents Strings as though they are primitives. But Strings are reference types!)

- You may use this plugin, but please remember that you need to understand and format all the types correctly when we ask you to draw an object diagram in a test.
Draw the object diagram for the following code:

```java
Counter c1 = new Counter();
Counter c2 = c1;
c1.increment();
c2 = new Counter();
```
What is `c1.count`, according to your diagram?

- 0
- 1
- 2
- null
- None of the above
Solution

Counter c1 = new Counter();
Counter c2 = c1;
c1.increment();
c2 = new Counter();
Notes About Objects

- Once you think about classes as specialized custom types. That is pieces of pre-defined data (aka fields) over which you can operate a given set of methods. It gets a lot easier to wrap your head around all the terms.
- Also the notion of Reference semantics is separate from that of classes/Objects. It’s just that Java chooses to implement Class Objects using reference semantics.
Value vs Reference Semantics

- Data of variable contained directly in the variable’s state
- All primitive types in java implement this.

- Variable contains the location (also called reference) for the associated state data of the variable
- All Class Types in Java implement this
Mutability

Variables
• By default, variables can be reassigned
  • `int x = 0;`  
    `x = x + 1;`

• A **final** variable can only be assigned to once
  • `final int serialNum = sn;`

Types
• Values of **immutable** types cannot change their *state*
  • Mimic value semantics
  • Variables change value through reassignment

• Values of **mutable** types can change their state
  • Side effect of methods (reassign or mutate fields)
  • Changes visible through all aliases
Example: immutable vs. mutable Point

class Point {
    final double x;
    final double y;
    double x() { return x; }
    double y() { return y; }
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    Point shifted(double dx, double dy) {
        return new Point(x + dx, y + dy);
    }
}

class MPoint {
    double x;
    double y;
    double x() { return x; }
    double y() { return y; }
    MPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
    void shift(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
Client code: immutable vs. mutable

```java
final Point p = new Point(1, 2);
Point q = p;
// Move point left
p = p.shifted(-4, 0);
```

```java
final MPoint p = new MPoint(1, 2);
MPoint q = p;
// Move point left
p.shift(-4, 0);
```
Designing Classes

- So far we’ve seen syntactically how classes operate in Java
  But the goal is to translate your ideas into Java!
  - Organize the logic of your program in terms of Objects interacting with one another in a program that gets executed.
    - For example, in a game, you can have multiple instances of game avatar Objects interacting with one another
What are some other real world problems of interest you can think of, that can be modeled as Objects interacting with one another?
How do we do this for any problem?

- Design classes (aka templates) for each of entities we want to model. Aka fields and methods.
- But what tells us how the fields should behave or how they should be used by a user of this class?
Invariants, constructors, getters and setters
Object state

• Recall: A **type** is a set of values…
  • Which values are allowed?

• Is a **Counter** object with counts=-1 valid?
• What about a dictionary that’s not in alphabetical order?
Class invariants

- Invariant: a statement that should always be true
  - **Class invariant**: relationship between fields; truthfulness not affected by calling methods
    - Example: 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
  - Can write a method to check them – catches bugs
Example: Counter

• State:
  \[ \texttt{int} \ \texttt{counts}; \]

• Allowed states:
  \([-2147483648, 2147483647]\]

• Which states \textit{should} be allowed?

• Can any behavior implementations yield disallowed states?
Example: Counter

• State:
  int counts;

• Allowed states:
  [-2147483648, 2147483647]

• Is this an invariant for class counter?
Types and invariants

• **Static typing** enforces common invariants automatically!
  • “`int` counts is an integer”
  • “`String` name is a String”

• If an invariant concerning a field is captured in the field’s type, you do not need to document it separately. It’s probably be present in Java’s documentation for the type and you are simply it’s user.
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?

• Note that there’s consequences to these design choices and its hard to get them perfectly right.
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
  • Initialize all fields to default values

```java
class Counter {
    int counts;
    Counter() {
        counts = 0;
    }
    // ...
}
```
Constructor syntax example

class Point {
    final double x;
    final double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

Point p = new Point(1, 2);
Example: Fraction

• What fields could represent a fraction?

• Are any field values invalid?

• If representing points on number line, are any field values redundant?
Encapsulation
Enforcing invariants

- Who is responsible for enforcing a class’s invariants?
  - The class implementer: responsible for the class’s .java file; must ensure every method preserves the invariant
  - What about users of the class (“clients”)? If they can overwrite fields, they can violate invariants

- Beyond invariants: abstraction & decoupling
  - What if implementer wants to change how state is represented?
  - What vs. how
    - Method signature and specification says “what”
    - Method body (relying on fields, hence class invariant) says “how”
Encapsulation

• Can a programming language 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

```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 == Integer.MAX_VALUE) { counts = 0; }
        else { counts += 1; }
    }
}
```
Encapsulation

• Procedural programming: data is “in the open”, accessible by code from anywhere in the program (written by different people)
  • Must trust all code to preserve invariants associated with all data
  • To change data layout, must update code everywhere

• OOP bundles data with associated behavior, prevents direct access
  • Only class implementer must be trusted to preserve invariants
  • Class implementer can change data representation without changing any code outside of class
  • Decouples code using an “abstraction barrier”
Anti-pattern: getters/setters

• Many Java classes mark their fields as private, then declare public “getter” and “setter” methods to provide read and write access
  • Wrong mentality: focus is on state rather than behavior
  • Think in terms of “observers”, not “getters”; no need for “get” prefix

• Encapsulation is about abstraction, not just protection
  • Best practice: choose behaviors before state representation
### All Of Java’s Main Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference/Value Semantics</td>
<td>Mutability/Immutability</td>
</tr>
<tr>
<td>Classes</td>
<td>Encapsulation</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Enables relationships between Objects/Classes. Ton’s of features result from this.</td>
</tr>
</tbody>
</table>

OOP

Basic Data semantics
Access modifiers

• **protected**: Accessible by subclasses (wait for inheritance)

• Default (nothing): Accessible within package

• Can make constructors private
  • Prevents users from using new-expressions