Recursion (cont.); Side effects

- Finish recursion (slides from last lecture)
- Review of encapsulation
- Side effects (topic leftover from before exam)
  - Methods with side-effects
    - changing object passed as explicit parameter
    - immutable classes
  - Return values
    - returning references from inside objects
  - Copying objects

Announcements

- Lab this week does not involve Vocareum (see lab description) [also no Vocareum link on d2l]

Recursion (cont.)

- See Tue. slides.

Review of Encapsulation

- Prompted by…
  - **Very lowwww** average score on MT1 Qn 4.2:

    What do programmers of client code have to do when implementor changes from your **`TimeOfDay`** Rep 1 to **`TimeOfDay`** Rep 2 and why?
  - Several recent piazza questions on PA2 confusing user interface with class interface, and error checking with isValid methods or assert failures.

Review: Some OO principles

- encapsulation of classes (aka, information hiding)
- splitting responsibilities between classes
- Why?
  - managing complexity of large programs
  - easier to develop and debug code (unit-testing)
  - easier to modify code (limit scope of changes)
  - reusable classes (less work next time)
- (adding more next time: inheritance)

Example: PA 2 Class Design: who knows what?

Avoiding Side effects

- Horstmann Section 8.2.4 discussed avoiding creating methods with side-effects.
- E.g.: changing *implicit* parameter (mutator) -- ok

```
    account.withdraw(amount)
```

- only change *explicit* param if it makes sense and is documented:

```
    account.transfer(amount, otherAccount);
```

- should be no surprises for client….

`gradeBook.addStudents(studentNamesArray);`

(by the way, empties out **`studentNamesArray`**. oops.)

Another kind of side-effect

- Save a reference to object passed to one method,
- But modified in another method.
- Example: Drunkard’s walk problem:

```
Drunkard d = new Drunkard(new Point(3, 7));
System.out.println(“Starts at: “
                          + d.getCurrentLoc());
for (int i = 0; i < 100; i++) {
   d.takeStep();
   System.out.println(“Moves to: “
                          + d.getCurrentLoc());
}
```

- Suppose the **`Drunkard`** class uses a Java **`Point`** to represent its current location.

Drunkard example (cont.)

```
public class Drunkard {
   private Point currentLoc; . . .
   Drunkard(Point startLoc) {
      currentLoc = startLoc;  . . .
   }
   void takeStep() { . . .
      currentLoc.translate(dx, dy);
   } . . .
}
```

```
Point startLoc = new Point(100, 100);
Drunkard d = new Drunkard(startLoc);
d.takeStep();      // suppose he moves to (100, 105)
System.out.println(startLoc + “ “ + d.getCurrentLoc());
```

POLL: Hint: draw a box and pointer diagram

Asynchronous participation: Link to Drunkard poll 1

Drunkard example (cont.)

```
public class Drunkard {
   private Point currentLoc; . . .
   Drunkard(Point startLoc) {
      currentLoc = startLoc;  . . .
   }
   . . .
}
```

```
Point startLoc = new Point(100, 100);
Drunkard d = new Drunkard(startLoc);
d.takeStep();      // suppose he moves to (100, 105)
System.out.println(startLoc + “ “ +
  d.getCurrentLoc());
```

Added more code to Ex.

```
public class Drunkard {
   Point currentLoc; . . .
   Drunkard(Point startLoc) {
       currentLoc = startLoc;  . . .
    }
    void takeStep() { . . .
       currentLoc.translate(dx, dy);
    } . . .
}
```

```
Point startLoc = new Point(100, 100);
Drunkard d = new Drunkard(startLoc);
d.takeStep();      // suppose he moves to (100, 105)
System.out.println(startLoc + “ “ + d.getCurrentLoc());

startLoc.translate(6, 12);
System.out.println(startLoc + “ “ + d.getCurrentLoc());
```

Asynchronous participation: Link to Drunkard poll 2

Solutions:

- pass in x, y separately instead.
  - Not great: lose **`Point`** abstraction.

- Better: Drunkard makes a *copy* of the Point object passed in:

```
public Drunkard(Point startLoc) {
   currentLoc = new Point(startLoc);
}
```

(called a defensive copy)

- Best: Use an immutable type for the contained object instead…

Immutable classes

- don't have to worry about side-effects when class is immutable:
- Reminder: has no mutators
- Safe to have multiple references to same object
- Good to make our own classes immutable, when it makes sense (e.g., **`ImPoint`**, **`Interval`** (F17, MT 1), **`Term`**)

Return values from methods

- Similar semantics to parameter-passing:
- everything is returned by value
- either a copy of a primitive value

```
double  xVal = point.getX();
```

- or a copy of an object reference: whole object is not copied:

```
public Point getCurrentLoc() {
  return currentLoc;
}
```

- What's the danger here?

Object references with return values

```
Drunkard d = new Drunkard(new Point(5, 10));
Point myPoint = d.getCurrentLoc();
d.takeStep();              //moves to (10,10)
System.out.println(myPoint + “ “ + d.getCurrentLoc());
myPoint.translate(100, 100);
System.out.println(myPoint + “ “ + d.getCurrentLoc());
```

Solution

- Make a copy when returning a reference to an object "owned" by the enclosing object.

```
Point getCurrentLoc() {
  return new Point(currentLoc);
}
```

- …unless the contained object is of an immutable type (e.g., safe to return **`ImPoint`**, **`String`** or **`Term`**)

- copying a passed parameter or value to be returned is called making a *defensive copy*

Who owns the contained object?

- **`Drunkard`** owns its **`currentLoc`**
  - **`Drunkard`** object is the only code that can modify it
- **`ArrayList`** doesn't own its elements
  - Container to organize them for the client
  - Client can mutate the elemts in the **`ArrayList`**:

```
ArrayList<Point> pointList;
   . . .
pointList.add(myPoint);  // doesn't copy myPoint
   . . .
pointList.get(0).translate(5,10);
```

When to copy vs. share objects?

- Sometimes want shared objects rather than copies: e.g., Two Computers share a Printer. (more examples coming in pa3)

- Do not have to do a defensive copy of a contained immutable object.

- Good to make classes that represent a value immutable. E.g., Point, Integer, Fraction, ComplexNumber, String