More on class design

- From last time:
  - finish discussing Representation Invariants
- What does a class represent?
- Minimizing inter-method dependencies
- Choosing instance variables
  - minimizing scope
- Review of copy semantics
- Parameter passing

Announcements

- Midterm 1 is on Thu 2/16
  - sample problems have been published
  - Location: THH 101
  - Closed book, closed note, no electronic devices
  - Bring USC ID card
- Don’t wait until after MT to start PA2
- Published completed code for **`Names`** example

  in **`~csci455/code/02-07/complete`**

Class is a single concept

- Class should represent a single concept
- An object in the real world
  - (or from math, or a software artifact)
- E.g., Point, Rectangle, Bar, Paycheck
  - Methods all relate to that single concept:
  - get info about the object (accessor)
  - manipulate the object (mutator)
- Can make multiple instances of the class

A bad class design

```
class MyProgAssgt {
    public void doStep1() { . . . }
    public void doStep2() { . . . }
    public void doStep3() { . . . }
    // instance variables are effectively
    // "global" vars
}
```

- Can you make multiple instances of the object?
- What is the data abstraction it represents?

Minimizing inter-method dependencies

- Generally want to be able to call methods in any order. e.g., Names: lookup, insert, remove
- Minimize the different states object can be in
- For implementor:
  - minimize instance variables to represent that state.
  - minimize different states of internal representation (avoids special-case code)

Some objects naturally have multiple states

- Have to think through what they are and transitions between them
- Ex: cash register class from Ch. 3 (and lab 3)

- We won’t encounter this much in CS 455.

Choosing instance variables

- For implementor: Instance variables are the input to every method.
- Need a clear understanding of what values are for, and how they are interrelated
- Suppose we had the following **`CoinTossSimulator`** instance variables. Which of them can we eliminate?

```
int totNumTrials;  // total since last reset
int currNumTrials; // total for this run
int numHeadsTails;
int numTailsTails;
int numHeadsHeads;
int i;             // which trial we are on
Random generator;
boolean doneReset; // have we done a reset?
```

A general principle:

- Minimize scope of variables / methods
  - public vs. private
  - instance var vs. local var

- Also one of our style guidelines for the class

Minimize scope: another example

- Proposed solution for reuse **`lookup`** code: Adding a data member so **`remove`** could use **`lookup`**:

```
class Names {
  private String[] namesArr;
  private int numNames;
  private int locFound;  // when is this init'd?
  . . .
  public boolean lookup(…) { … locFound = . . .  }
  public boolean remove(…) {
    . . . lookup(…);
    i = locFound; . . .
  }
  . . .

}
```

- Is **`locFound`** initialized when we enter **`lookup`**? **`remove`**? **`insert`**?
- If only used within **`remove`**, then should be local.

Second example (cont.)

- Reminder: improved solution
- private helper method

```
class Names {
   private String[] namesArr;
   private int numNames;
   ~~private int locFound;~~
   . . .
   public boolean lookup(…) { …lookupLoc(…)… }
   public boolean remove(…) {…lookupLoc(…)… }
   private int lookupLoc(…) {  }
   . . .
}
```

Choosing instance variables (cont.)

- Scenario: use an **`ArrayList`** representation for **`Names`** class.
- Suppose we had the following **`Names`** instance variables:

```
ArrayList<String> namesArr;
int numNames;
```

- Why is this not ideal?

Review of instance variables

- For implementer: Instance variables are the input to every method.
  - want to minimize how many
  - and how many different states they can be in
- Need a clear understanding of what values are for, any restrictions on them, and how they are interrelated
- Explicit statement of the last two is the

  representation invariant

Review of copy semantics: primitives

- Primitive types have *value semantics*

```
    int i = 0;
    int j = 3;
    i = j;
```

Review of copy semantics: objects

- Object and array types have *reference semantics*

```
Rectangle r = new Rectangle();
Rectangle t = new Rectangle(5, 5, 5, 5);
r = t;
r.translate(10, 10);
t = null;
```

Review of copy semantics: arrays

- Object and array types have *reference semantics*

```
int[] iArr = new int[5];
int[] jArr = new int[3];
iArr = jArr;
```

Review of copy semantics: immutable object types

- E.g., **`String, Term, Polynomial`**
- Can treat as if value semantics – but still have to create the object:

```
Polynomial p = null;
p = new Polynomial(new Term(3,2));
Polynomial q = p;
q.add(q);
p = q.add(q);
```

Parameter passing in Java

- All Java parameters are passed by value.

- Value and reference semantics also apply to parameter-passing rules:
  - Primitive types use value semantics
  - Object types (and arrays) use reference semantics

- Let’s see what this means . . .

Parameter passing in Java: primitive types

- all parameters passed by value. E.g.,

```
public static void foo(int x) {
    x = 0;
}
```

has no effect on caller:

```
int y = 10;
foo(y); // y unchanged
```

Parameter passing: object references

- for objects, the object *reference* is passed by value. E.g.

```
public static void foo(BankAccount account) {
   account = null;
}
```

has no effect on caller:

```
BankAccount myAccount = new BankAccount(100);
foo(myAccount);
myAccount.getBalance();  // 100
```

Passing object references by value

- method can't change *which* object `myAccount` refers to
- But it could still change what's *inside* the object
  by calling one of its methods:

```
public static void evil(BankAccount account) {
  account.withdraw(account.getBalance());
}
```

- Call:

```
BankAccount myAccount = new BankAccount(100);
evil(myAccount);
int bal = myAccount.getBalance();
```

How to “change” a primitive var in a method

Can *use return value* to update a single variable:

```
 public static int incr(int x) {
   return x+1;
 }
```

Sample call:

```
int x = 5;
x = incr(x);
```

Similar idea with immutable object:

```
Polynomial p = new Polynomial(…);
p = p.add(p);
```

Example: *cannot* write a swap method in Java

Method definition:

```
 public static void swap(int x, int y) {
   int temp = x;
   x = y;
   y = temp;
 }
```

Sample call:

```
int a = 5;
int b = 10;
swap(a, b);
```