Lecture 18-19: Aliasing, Mutability, Floating Point Calculations, Rational Numbers

Last time:

1. Unit testing and JUnit
2. Constructors revisited
3. equals

Today:

1. Project #4 assigned
2. Aliasing and Mutability
3. Floating Point calculations
4. Example class development: Rational Numbers

CMSC 131 Spring 2007
Bonnie Dorr (adapted from Rance Cleaveland)

Project #4 Is Assigned

- It is due Friday, 3/16 at 11 pm
- The project is closed
  - You must complete the project by yourself
  - Assistance can only be provided by teaching assistants (TAs) and instructors
  - You must not look at other students' code
- Start now!
  - Read entire assignment from beginning to end before starting to code
  - Check out assignment now from CVS
  - Follow the instructions *exactly*, as much of grading is automated

What about Strings and Aliasing?

- `String` objects are *immutable*; fields cannot be changed once created
  - Mutable objects: fields (values of instance variables) can be changed (e.g. Cat, Student, via `set` methods, etc.)
  - Immutable objects: fields (values of instance variables) cannot be changed
  - See `String` API: http://java.sun.com/j2se/1.3/docs/api/java/lang/package-summary.html

- In this example, the original string is not modified:

```
String x = "copy";
x += "cat";
```

  The variable `x` points to the new `"copycat"` string, but the original `"copy"` String object is not changed!

- Aliasing is not a problem with `String` objects because of immutability

Mutability: Return to the Cat Example

```
public Cat(Cat otherGuy) {
  name = otherGuy.name;
  lives = otherGuy.lives;
}
```

We add the above copy constructor to the Cat class.

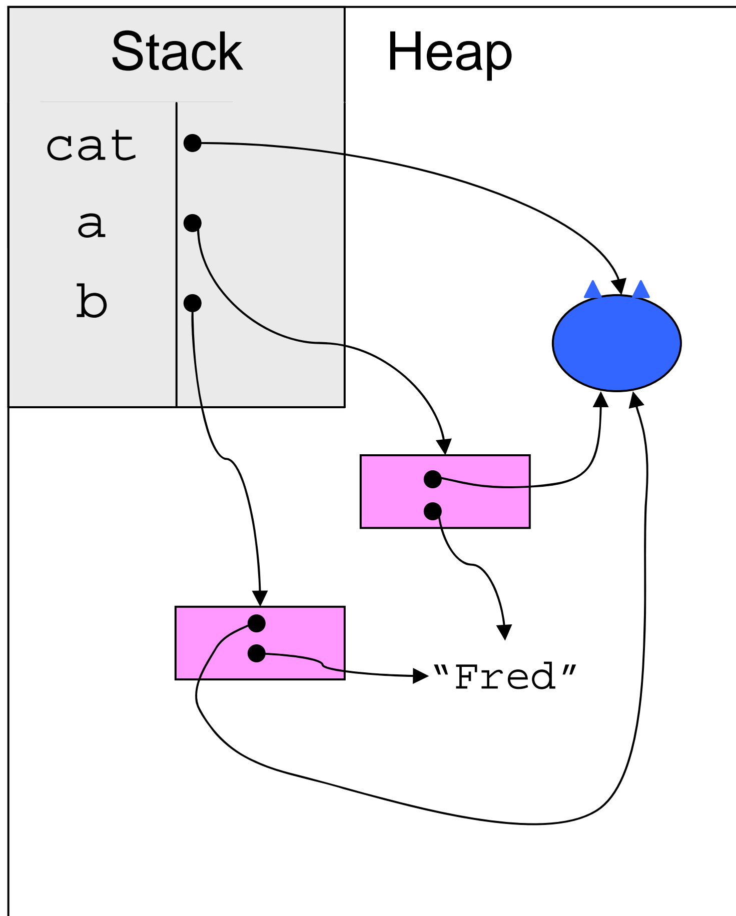
Mutability (continued)

… And we create a new CatOwner class:

```
public class CatOwner {

    private String name;
    private Cat pet;

    public CatOwner(String name, Cat pet) {
          this.pet = pet;
          this.name = name;
    }

    public CatOwner(CatOwner otherGuy) {
          pet = otherGuy.pet;
          name = otherGuy.name;
    }

    public static void main(String[] args) {
          Cat cat = new Cat();
          CatOwner a = new CatOwner("Fred", cat);
          CatOwner b = new CatOwner(a);
    }

}
```

Note that the CatOwner's copy constructor does not make a copy of otherGuy's pet! So the new CatOwner's pet will point to the same Cat as the otherGuy's pet

What does the stack/heap look like just after the last statement of main method in CatOwner class?



Aliasing:

- Two cat owners share the same cat!
- Two cat owners share the same name!
- Which of these two "sharings" (aliases) are bad?
  - The ones where a ***mutable*** object (cat) is being shared. (The values of the cat's instance variables may be changed!)
- Is it okay to share the string "Fred"?
  - Yes! Why? Because a String is an ***immutable*** object.
- Let's fix the aliasing involving the mutable object …

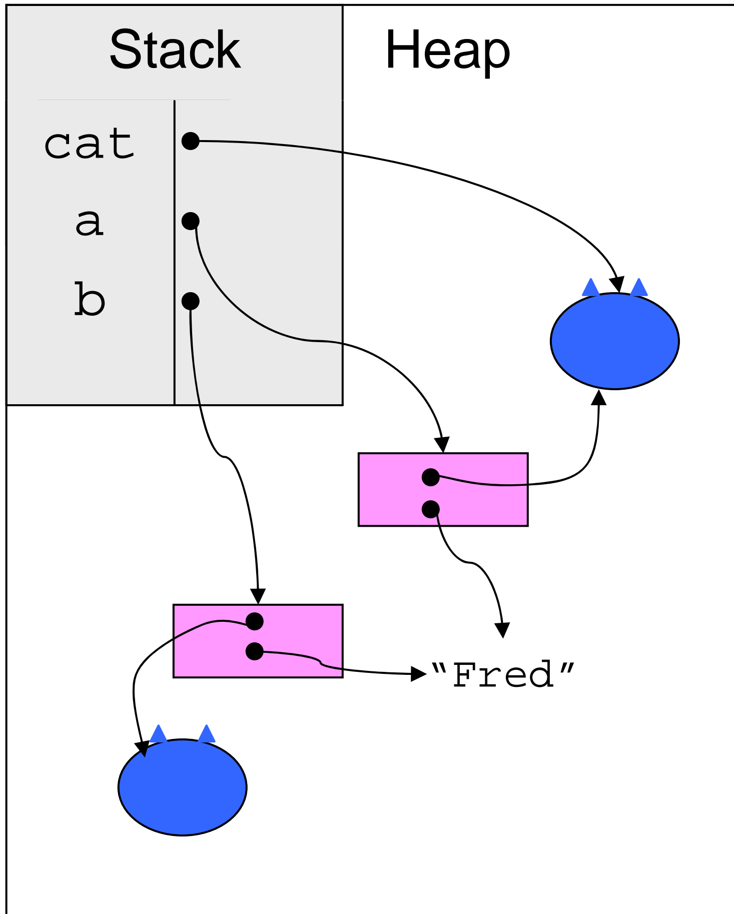
Mutability (continued)

… Modify CatOwner copy constructor:

```
public CatOwner(CatOwner otherGuy) {
    pet = new Cat(otherGuy.pet);
    name = otherGuy.name;
}
```

Now what does the memory map look like?

Memory map (with Cat copy constructor usage) just after last statement at end of main method in CatOwner class.



Aliasing:

- Now each catOwner has their own cat.
- Two cat owners still share the same name.

Floating Point Calculations

What will this print?

```
public class SimpleMath {
   public static void main(String[] args) {
        if (3.9 - 3.8 == 0.1) {
                System.out.println("I am a very smart computer.");
        } else {
                System.out.println("I can't do simple arithmetic.");
        }
   }
}
```

→ I can’t do simple arithmetic.

- Why?
- Conversion of floating point to binary leads to precision errors!
- What can we do?

Floating Point Calculations (cont.)

Two important rules:

- You can never use == to compare floating point values. Instead, check if two numbers are within a certain tolerance of each other.
- Never use floating point values to represent money, e.g., 3.52 to represent $3.52. Instead, use integer 352 to represent 352 pennies.

Revised Floating Point calculations

```
public class SimpleMath {
  final EPSILON = 0.0000000001;
  public static void main(String[] args) {
     if (Math.abs((3.9 - 3.8) - 0.1) < EPSILON) {
          System.out.println("I am a very smart
  computer.");
     } else {
          System.out.println("I can't do simple
  arithmetic.");
     }
  }
}
```

→ I am a very smart computer.

Today we will start an extended example

- We will implement a class, Rational, for (immutable) rational numbers
- The class will include
  - Constructors
  - Arithmetic operations (+, -, *, /)
  - `toString`
  - Comparisons (`equals`, `compareTo`)

Rational Numbers?

- Fractions!
  e.g.: 3/4, 15/8, -1/7, etc.
- Conventions
  - Integers represented as follows: 7/1, 0/1
  - Numerator, denominator are in lowest terms:
    e.g. 2/3 rather than 4/6
  - Numerator can be negative, but denominator should be positive
  - 0 not allowed in a denominator

Two Simple Constructors for Rational

- `Rational (int n, int d)`

  Representation of n/d
- `Rational (int n)`

  Representation of n/1

Rational Class: Two Constructors

```
public class Rational {
  private int num, den;

  public Rational(int num, int den) {
      this.num = num;
      this.den = den;
      }

  public Rational(int num) {
      this.num = num;
      den = 1;
      }
}
```

“Lowest Terms”?

- How do we represent the fraction 20/60 ?
  - Reduce to lowest terms.
- Given a fraction p/q, how do you put it into lowest terms?
- Method
  - Find greatest common divisor (gcd) of p, q
    - gcd of p, q: largest number that divides both p, q
    - Euclid’s algorithm (beyond scope of this lecture) performs this if p, q are both positive
  - Replace p/q by (p/gcd) / (q/gcd)
- Example
  - Consider 18/24
  - gcd of 18, 24 is 6
  - So 18/24 = (18/6) / (24/6) = 3/4

Reducing a Rational to Lowest Terms

Idea: Find the minimum of the numerator and denominator. Count down from that lower number until a number that divides both the numerator and denominator is found (or the number 1 is reached). End the loop after dividing both the numerator and denominator by that number.

```
private void reduce() {
   int smaller = Math.min(num, den);
   boolean done = false;
   for (int i = smaller; (i>=2 && !done); i--) {
        if (num % i == 0 && den % i ==0) {
                num /=i;
                den /=i;
                done = true;
        }
   }
}
```

Question: Where do we call reduce() ?

→ Inside the constructor(s).

Adding reduce() to constructors

```
public Rational(int num, int den) {
    this.num = num;
    this.den = den;
    reduce();
    }

public Rational(int num) {
    this.num = num;
    den = 1;
    reduce();
    }
```

Hints

- Come up with representative test cases
- Intertwine implementation and testing
  - Do constructors and getters first, then test
  - Implement “related operations”, then test
- Rerun each test (even ones for previously tested methods) when you test
  - This is called regression testing
  - Useful for detecting changes that may invalidate previous test results!
  - Easy to set up in Eclipse
- Use debugger to track down sources of errors in tests

Apply Different Test Cases for Rational Numbers

```
public static void main(String[] args) {
    Rational r = new Rational(1,3);
    System.out.println(r.num);
    System.out.println(r.den);

    r = new Rational(35,25);
    System.out.println(r.num);
    System.out.println(r.den);

    r = new Rational(60,20);
    System.out.println(r.num);
    System.out.println(r.den);

    r = new Rational(25,35);
    System.out.println(r.num);
    System.out.println(r.den);

    r = new Rational(20,60);
    System.out.println(r.num);
    System.out.println(r.den);
}
```

- Test case for reduced rational number
- Test case where num is higher and num does not divide den (nor vice versa)
- Test case where num is higher and den divides num
- Test case where den is higher and num does not divide den (nor vice versa)
- Test case where den is higher and num divides den