Objectives

- Liskov Substitution Principle
- Refactoring for Extensibility

Review

- What are reasons that we refactor our code?
  - What are the goals in refactoring?
  - What is not necessarily a goal?
- What are some clues we can use that we should consider refactoring our code?
  - What are those clues called?

Refactoring Practice

Bookstore application

```
if (type.equals("topic")){
   String array[] = new String[2];
   for (int i = 0; i < bookNum; i++){
       Book currentBook = books.get(i);
       if (currentBook.getTopic().equals(argument)){
           flag += 1;
           array[0] = "true";
           if (flag == 1) //if we append each time, there will
                      // be a 'null' at the beginning
              array[1] = "Title: " + currentBook.getTitle() + "\n"
                     "Item Number: " + currentBook.getItemNumber();
           else array[1] += "Title: " + currentBook.getTitle() +
              "\n" + "Item Number: " + currentBook.getItemNumber();
       }
   }
   if (flag == 0){
       array[0] = "false";
       array[1] = "No items match your query\n";
   }
}
```

Refactoring Practice

Refactoring is a general concept: applies to Python too!

```
if prompt.split(" ")[0] == "buy" and nextCond == False:
    result = server.FrontEnd.buy(int(prompt.split(" ")[1]))
    array = server.FrontEnd.lookup(int(prompt.split(" ")[1]))

    if result == True:
         nextCond = True
         print "bought book " + str(array[0])

    else:
         nextCond = True
         print "Book out of stock or doesn't exist"
```

Refactoring Practice

Refactoring is a general concept: applies to Python too!

```
values = prompt.split(" ")
if values[0] == "buy" and not nextCond:
    myVal = int(values[1])
    result = server.FrontEnd.buy(myVal)
    array = server.FrontEnd.lookup(myVal)

    if result:
         nextCond = True
         print "bought book " + str(array[0])

    else:
         nextCond = True
         print "Book out of stock or doesn't exist"
```


LISKOV SUBSTITUTION PRINCIPLE

Liskov Substitution Principle (LSP)

- The substitution principle:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

- In other words...

> If a module is using a base class, then it should be able to replace the base class with a derived class *without affecting the functioning of the module.*

 Liskov & Wing, 1994

Code Smell: Using `instanceof`

```
public void drawShape( Shape shape ) {
    if ( shape instanceof Square ) {
        drawSquare(shape);
    }
    else if( shape instanceof Circle ) {
        drawCircle(shape);
    }
}
```

- Why isn't this good code?
- How could we write this in a better way?

Code Smell: Using `instanceof`

- Previous example: had to know all of the Shape classes
  - Update whenever a Shape is added or removed
- Better code: ***Polymorphic***!

```
public void drawShape( Shape shape ) {
    shape.draw();
}
```


Design by Contract

- Methods of classes declare preconditions and postconditions
  - Preconditions must be met for method to execute
  - After executing, postconditions must be true
    - Example for Rectangle's `setWidth`:
      - `myWidth == newWidth && myHeight == oldHeight`

Design by Contract and LSP

- Methods of classes declare preconditions and postconditions
  - Preconditions must be met for method to execute
  - After executing, postconditions must be true
    - Example for Rectangle's `setWidth`:
      - `myWidth == newWidth &&`
        `myHeight == oldHeight`
- For derivatives
  - Preconditions can only be weakened
  - Postconditions can only be strengthened
  - ➥ Derivatives must adhere to constraints for base class

Design by Contract and LSP

- Recall: User interacts with interface, e.g., the base class

What if preconditions are stronger?
What if postconditions are weaker?

- For derivatives
  - Preconditions can only be weakened
  - Postconditions can only be strengthened
  - ➥ Derivatives must adhere to constraints for base class

Summary of LSP

- Liskov Substitution Principle (a.k.a. design by contract) is an important feature of programs that conform to the Open-Closed Principle
- Derived types must be completely substitutable for their base types
- Derived types can then be modified without consequence

Rectangle Class

```
public class Rectangle {
   private int myHeight;
   private int myWidth;

   public void setWidth( int w ) {
      myWidth = w;
   }

   public void setHeight( int h ) {
      myHeight = h;
   }

   // getters…
}
```

Square Class

- A square is a rectangle
  - But a rectangle is not a square
- In the interest of code reuse

```
public class Square extends Rectangle
```

- Any problems with this implementation?
  - Inherits:

```
private int myHeight;
private int myWidth;
public void setWidth( int w );
public void setHeight( int h );
```


To Keep Square Consistent…

```
public void setWidth( int w ) {
    super.setWidth(w);
    super.setHeight(w);
}

public void setHeight( int h ) {
    super.setWidth(h);
    super.setHeight(h);
}
```

But What About Users of Classes?

- Consider the function:

```
public void testMethod( Rectangle r ) {
    r.setWidth(5);
    r.setHeight(4);
    assertEquals(20, r.getWidth()*r.getHeight());
}
```

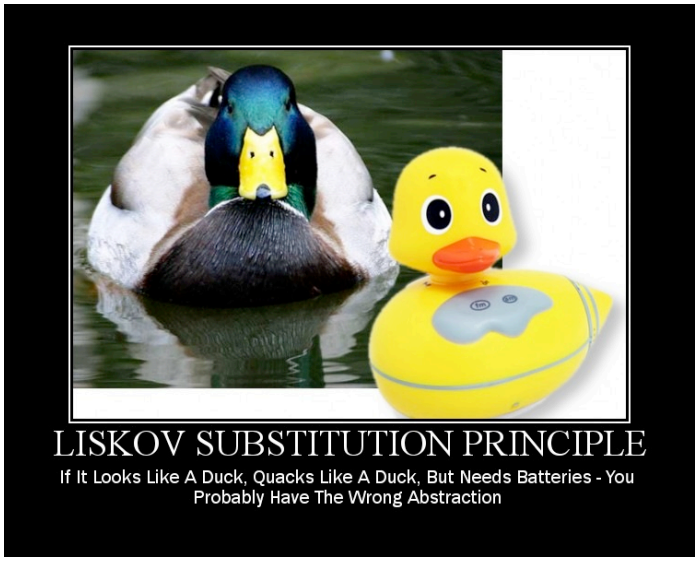
- What happens if method is called with a `Square` object?

The Problem

- A `Square` object is ***not*** a `Rectangle` object
- Behaviors are different
- Clients depend on behaviors

Lesson: All derivatives of class **must** have the same ***behavior***

http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/



Liskov Substitution Principle (LSP)

- Named after Barbara Liskov
  - MIT Professor of Engineering
  - 2008 ACM Turing Award
  - Contributions to programming languages, pervasive computing
  - Trivia: first woman in the United States to receive a Ph.D. from a computer science department (Stanford, 1968)

We have an advanced lab machine named after her.

 Liskov & Wing, 1994

& Wing

- Jeannette Wing
  - Corporate Vice President of Microsoft Research
  - Big proponent of computational thinking as assistant director for Computer and Information Science and Engineering at the NSF from 2007 to 2010.

Discussion of Abstraction

- What does abstraction allow?
- Are there any limitations to abstraction?

Summary of Designing for Change

Use ***abstraction*** for code that is *likely to change*

- Can depend on code that is *stable* and unlikely to change
  - Example of stable code: `System.out`

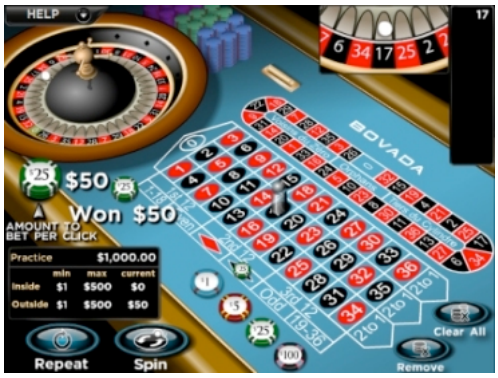
Refactoring Summary

- Write code and then *rewrite* code
  - Eye toward extensibility, flexibility, maintainability, and readability
  - Maintain correctness
- Reading/understanding other people's code can be difficult
  - Make your code readable, understandable
- Probably impossible to design/write "correctly" the first time
  - A lot harder to get the logic right, make sure you're not creating bugs, know/check the right answer…
  - Could cause yourself headaches coding this way first

REFACTORING FOR EXTENSIBILITY

Simulating a Roulette Game

- See handout



In Eclipse, Import Existing Project into Workspace
`roulette.tar` on the course web site

Understanding Code

- Execute the code
  - What is the main driver for this project?
- What are each class's responsibilities?

Bug in the Code

- Determining if Odd/Even Bet was won is incorrect

Understanding Code

- Focus: how **open** is the code to adding **new** kinds of **bets** and how **closed** it is to **modification**?
  - How many classes know about the `Bet` class?
  - What code would need to be added to `Game` to allow the user to make another kind of bet that paid one to one odds and was based on whether the number spun was high (between 19 and 36) or low (between 1 and 18)?

Roulette

- Goals
  - Learn to read, understand someone else's code
    - Refactoring can help
  - Refactor for readability
  - Justify decisions
- No "right" answer
  - Many design decisions
  - Want you to defend your design decision in code critique

TODO: Assign 8

- For Wednesday:
  - Read through the Roulette code
  - Answer the questions under understanding the design and refactoring
- For next Monday:
  - Refactoring code for extensibility
    - Discuss tradeoffs in designs
  - Testing!