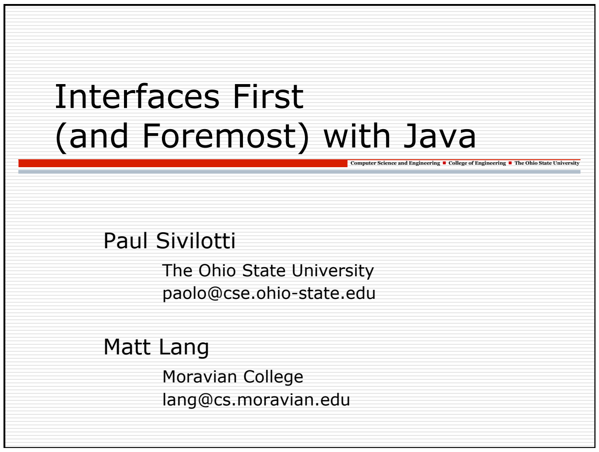
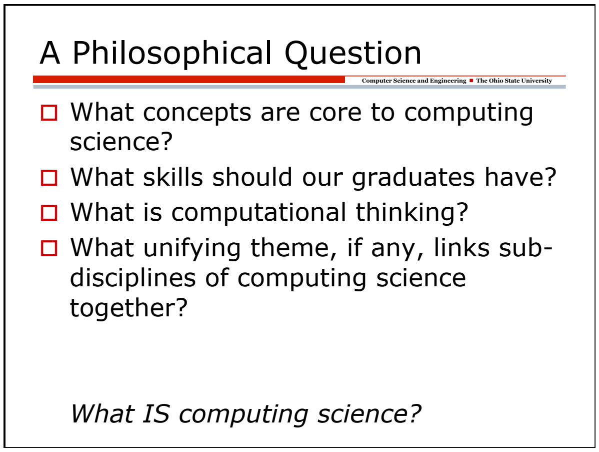


Interfaces First (and Foremost) with Java

Paul Sivilotti
The Ohio State University
paolo@cse.ohio-state.edu

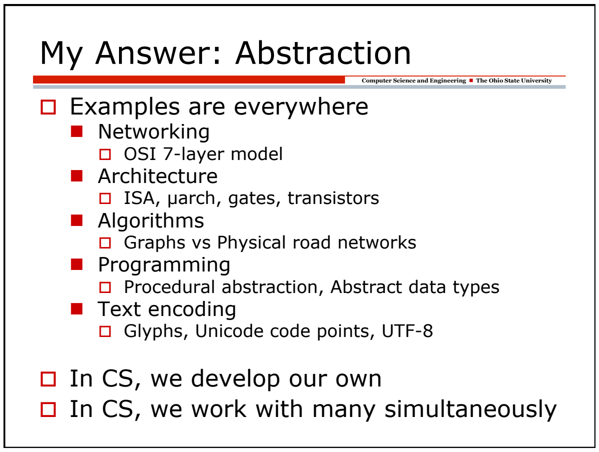
Matt Lang
Moravian College
lang@cs.moravian.edu



A Philosophical Question

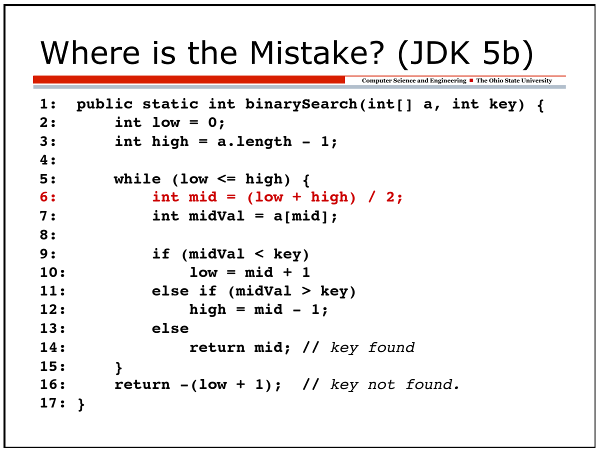
- What concepts are core to computing science?
- What skills should our graduates have?
- What is computational thinking?
- What unifying theme, if any, links sub-disciplines of computing science together?

What IS computing science?



My Answer: Abstraction

- Examples are everywhere
  - Networking
    - OSI 7-layer model
  - Architecture
    - ISA, μarch, gates, transistors
  - Algorithms
    - Graphs vs Physical road networks
  - Programming
    - Procedural abstraction, Abstract data types
  - Text encoding
    - Glyphs, Unicode code points, UTF-8
- In CS, we develop our own
- In CS, we work with many simultaneously

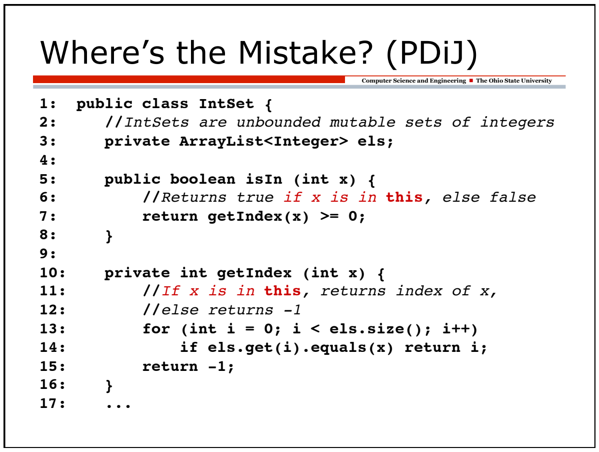


Where is the Mistake? (JDK 5b)

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

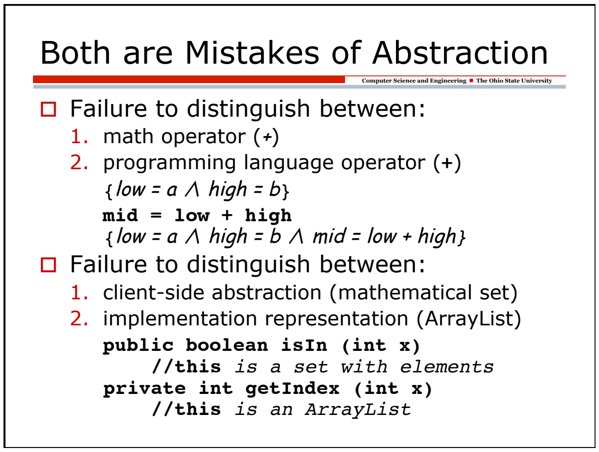


Where's the Mistake? (PDiJ)

```
public class IntSet {
   //IntSets are unbounded mutable sets of integers
   private ArrayList<Integer> els;

   public boolean isIn (int x) {
       //Returns true if x is in this, else false
       return getIndex(x) >= 0;
   }

   private int getIndex (int x) {
       //If x is in this, returns index of x,
       //else returns -1
       for (int i = 0; i < els.size(); i++)
           if els.get(i).equals(x) return i;
       return -1;
   }
   ...
```



Both are Mistakes of Abstraction

- Failure to distinguish between:
  1. math operator ($+$)
  2. programming language operator (+)

  $\{low = a \wedge high = b\}$

  mid = low + high

  $\{low = a \wedge high = b \wedge mid = low + high\}$
- Failure to distinguish between:
  1. client-side abstraction (mathematical set)
  2. implementation representation (ArrayList)

```
public boolean isIn (int x)
    //this is a set with elements
private int getIndex (int x)
    //this is an ArrayList
```

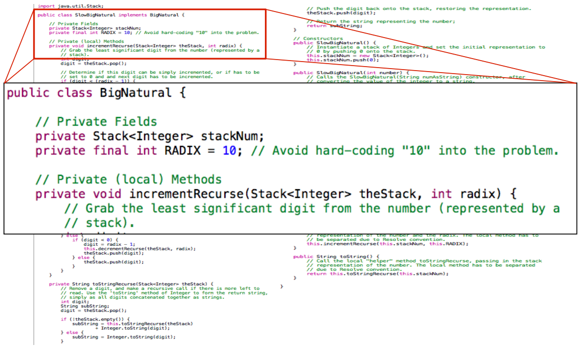
An OO Course

- Variables, assignments, conditionals
- Iteration
- Objects: Classes vs instances
- State and behavior: Fields vs methods
- Encapsulation: Private vs public
- Inheritance

Example: Natural Numbers

- Write a Java class that represents unbounded natural (ie >= 0) numbers
  - Like BigInteger, but for natural numbers
- Requirements:
  - Two methods: increment and decrement
  - Increment increases the value by 1
  - Decrement decreases the value by 1, unless it is already 0, in which case it leaves the value unchanged

A Solution



```
import java.util.Stack;
public class SlowBigNatural implements BigNatural {

    // Private Fields
    private Stack<Integer> stackNum;
    private final int RADIX = 10; // Avoid hard-coding "10" into the problem.

    // Private (local) Methods
    private void incrementRecurse(Stack<Integer> theStack, int radix) {
        // Grab the least significant digit from the number (represented by a
        // stack).
        int digit;
        digit = theStack.pop();

        // Determine if this digit can be simply incremented, or if has to be
        // set to 0 and and next digit has to be incremented.
        if (digit < (radix - 1)) {
```

```
        // Push the digit back onto the stack, restoring the representation.
        theStack.push(digit);

        // Return the string representing the number;
        return subString;
    }

    // Constructors
    public SlowBigNatural() {
        // Instantiate a stack of Integers and set the initial representation to
        // 0 by pushing 0 onto the stack.
        this.stackNum = new Stack<Integer>();
        this.stackNum.push(0);
    }

    public SlowBigNatural(int number) {
        // Calls the SlowBigNatural(String numAsString) constructor, after
        // converting the value of the integer to a string.
```

```
public class BigNatural {

    // Private Fields
    private Stack<Integer> stackNum;
    private final int RADIX = 10; // Avoid hard-coding "10" into the problem.

    // Private (local) Methods
    private void incrementRecurse(Stack<Integer> theStack, int radix) {
        // Grab the least significant digit from the number (represented by a
        // stack).
```

```
        } else {
            if (digit < 0) {
                digit = radix - 1;
                this.decrementRecurse(theStack, radix);
                theStack.push(digit);
            } else {
                theStack.push(digit);
            }
        }
    }

    private String toStringRecurse(Stack<Integer> theStack) {
        // Remove a digit, and make a recursive call if there is more left to
        // read. Use the 'toString' method of Integer to form the return string,
        // simply as all digits concatenated together as strings.
        int digit;
        String subString;
        digit = theStack.pop();

        if (!theStack.empty()) {
            subString = this.toStringRecurse(theStack)
                    + Integer.toString(digit);
        } else {
            subString = Integer.toString(digit);
        }
```

```
        // representation of the number and the radix. The local method has to
        // be separated due to Resolve convention.
        this.incrementRecurse(this.stackNum, this.RADIX);
    }

    public String toString() {
        // Call the local "helper" method toStringRecurse, passing in the stack
        // representation of the number. The local method has to be separated
        // due to Resolve convention.
        return this.toStringRecurse(this.stackNum);
    }
}
```

Information Hiding vs Abstraction

- Information Hiding is:

```
class BigNatural {

        public void increment();


        public void decrement();
}
```

Information Hiding vs Abstraction

- Abstraction is:

```
//A BigNat is a non-negative unbounded
//integer
class BigNatural {

    //this = #this + 1
    public void increment();


    //if #this > 1, this = #this – 1
    //          else, this = 0
    public void decrement();
}
```

Our Approach: Interfaces

- Require every component to have *both*
  1. An interface, and
  2. A class implementing that interface
- The interface is a client-side (abstract) description of behavior
  - State given as fields of mathematical types
  - Methods with specifications in terms of abstract state
- Separate lexical scope enforces distinction

Information Hiding vs Abstraction

- Abstraction is:

```
//@mathmodel n is an unbounded integer
//@constraint n >= 0
//@initially constr() ensures n=0
interface BigNatural {
    //@alters this.n
    //@ensures n = #n + 1
    public void increment();

    //@alters this.n
    //@ensures if #n > 1, n = #n – 1
    //         else, n = 0
    public void decrement();
}
```

How do You Use Interfaces?

- Motivation: Java has single inheritance
  - Interfaces allow multiple "is a" relationships
- Motivation: Multiple implementations
  - Interfaces provide flexibility to choose different implementations
- Motivation: call-backs
  - Swing needs them

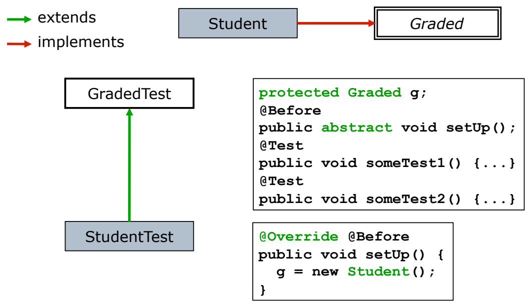
Outline of the Talk

- Motivation: Centrality of abstraction
- Abstraction ≠ private + getters/setters
- Take-home message:
  - Leverage separation enforced by interfaces
  - Require students to use/write/document both an interface and a class for each component
- Benefits
- Limitations

Benefit 1: Javadoc the Contract

- Best practice: Javadoc should describe *behavior* but not implementation details
- Tension: Javadoc for private methods?
  - Javadoc is standard documentation tool
  - Private fields and methods not part of the contract
- Interface+Class discipline resolves this tension
  - Javadoc everything in interface for clients
  - Javadoc everything in class for coders

Benefit 2: Blackbox JUnit Testing

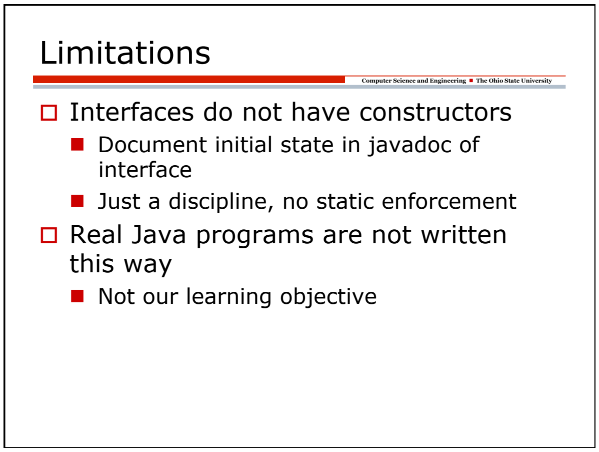


```
protected Graded g;
@Before
public abstract void setUp();
@Test
public void someTest1() {...}
@Test
public void someTest2() {...}
```

```
@Override @Before
public void setUp() {
  g = new Student();
}
```

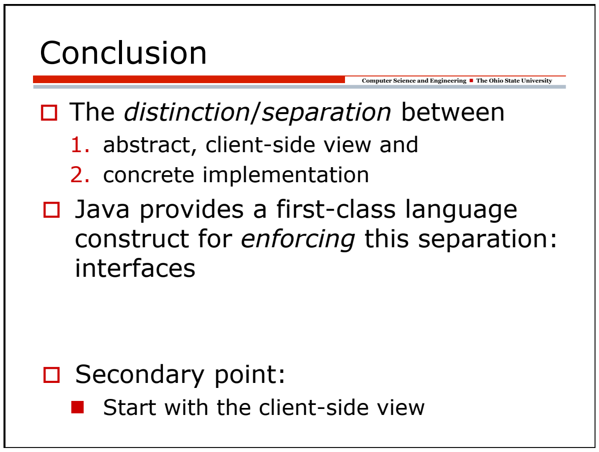
Other Benefits

- Behavioral subtyping
  - Class inheritance entails code sharing and overriding
  - Interface inheritance entails only behavioral refinement (ie subtyping)
- Designing exceptions
  - Exceptions must make sense in the interface
    - Eg ArrayIndexOutOfBoundsException reveals too much information about internal implementation
- Effective Java Item 52: Code to the interface



Limitations

- Interfaces do not have constructors
  - Document initial state in javadoc of interface
  - Just a discipline, no static enforcement
- Real Java programs are not written this way
  - Not our learning objective



Conclusion

- The *distinction*/*separation* between
  1. abstract, client-side view and
  2. concrete implementation
- Java provides a first-class language construct for *enforcing* this separation: interfaces
- Secondary point:
  - Start with the client-side view