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, parch, 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)

```java
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)

```java
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 (+)
     
`\( \text{low} = a \land \text{high} = b \land \text{mid} = \text{low} + \text{high} \)`

- Failure to distinguish between:
  1. client-side abstraction (mathematical set)
  2. implementation representation (ArrayList)
     
`\text{public boolean isIn (int x) \{ \}}`

```java
public boolean isIn (int x) {
    //this is a set with elements
    private int getlndex (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 (i.e., >= 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

Information Hiding vs Abstraction

- Information Hiding is:

```java
public class BigNatural {
    // Private Fields
    private Stack<Integer> stack;
    private final int BASE = 10; // Avoid hard-coding "10" into the problem.
    // Private (local) Methods
    private void increment() {
        // Add the least significant digit to the number (represented by a stack).
        // this = #this + 1
        public void increment();
    }
    public void decrement();
}
```

- Abstraction is:

```java
// 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:

```java
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

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

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
  - E.g. 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