Reduction, Abstract Data Types

\[ a \rightarrow \text{hypotenuse} \rightarrow h \]

\[
\text{double hypotenuse (double a, double b) \{ }
\]
\[
\text{    return Math.sqrt (a*a + b*b);} \]
\[
\text{}}
\]

procedural abstraction

\[ h = \sqrt{a^2 + b^2} \]
```java
double length(int x1, int y1, int x2, int y2) {
    int a = x2 - x1;
    int b = y2 - y1;
    return Math.sqrt(a*a + b*b);
}

double length(int x1, int y1, int x2, int y2) {
    return hypotenuse(x2-x1, y2-y1);
}
```
Rule 1: Computer Scientists are lazy. Reduction lets you reuse a slightly different solution.
Abstract Data Types (ADT) = interface + set of legal behaviors

You implement an ADT as a class

Class = data + methods (internal representation)

- Abstraction barrier
- Encapsulation = hiding/protecting the internal representation (the "rep")
Example: Rational Number ADT (immutable)

- **Constructor**
  - method name: `new`, `this`, `this.r
- new rational
- numerator, denominator
- rational number
- minus
- times
- divide
- invert
- negate
- plus
- minus


To String
Using `toString`

```java
System.out.println(new Rational(2, 3));
```

Output: 2/3

The rep:

```
<table>
<thead>
<tr>
<th>numer</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>denom</td>
<td>int</td>
</tr>
</tbody>
</table>
```
public class Rational {
    private int n;    // numerator
    private int d;    // denominator

    // constructor
    public Rational(int numer, int denom) {
        n = numer;       // reduce fraction?
        d = denom;
    }

    // for now, ignore the problem of zero denom.

    // To use: new Rational(3, 4)
public Rational plus(Rational r) {
    return new Rational((n*r.d + r.n*d, d*r.d));
}

\[
\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1d_2 + n_2d_1}{d_1d_2}
\]

\text{want}