Recitation 6

Loop Invariants and Prelim Review
Four loopy questions

// Precondition
Initialization;
// invariant: P
while ( B ) { S }

1. Does it **start** right?
   Does initialization make invariant P true?

2. Does it **stop** right?
   Does P and !B imply the desired result?

3. Does repetend S make **progress** toward termination?

4. Does repetend S **keep** invariant P true?
**Add elements backward**

<table>
<thead>
<tr>
<th>Precondition</th>
<th>b</th>
<th>?</th>
</tr>
</thead>
</table>

| Postcondition    | b | s = sum of these |

Get invariant by generalizing pre- and post-conditions

<table>
<thead>
<tr>
<th>Invariant</th>
<th>b</th>
<th>?</th>
<th>s = sum of these</th>
</tr>
</thead>
</table>
Add elements backward

```java
int s = 0;
int h = b.length - 1;
while (h >= 0) {
    s = s + b[h];
}
```

INV: \( s = \text{sum of } b \)

1. Does it start right?
2. Does it stop right?
3. Does it keep the invariant true?
4. Does it make progress toward termination?
Add elements backward

```java
int s = 0;
int h = b.length - 1;
while (h > 0) {
    s = s + b[h];
    h--;
}
```

**Loop Invariants**

INV: $b$  

| 0 | ? | $s = \text{sum}$ |

1. Does it **start** right? (Wrong)
2. Does it **stop** right? (Wrong)
3. Does it **keep** the invariant true?
4. Does it make **progress** toward termination?
Add elements backward

```
int s = 0;
int h = b.length-1;
while (h >= 0) {
    s = s + b[h];
    h = h - 2;
}
```

Loop Invariants

0. Does it start right?
1. Does it **stop** right?
2. Does it **keep** the invariant true?
3. Does it make **progress** toward termination?
Add elements backwards

```java
int s = 0;
int h = b.length - 1;
while (h >= 0) {
    s = s + b[h];
    h--;
}
```

**Loop Invariants**

1. Does it **start** right?
2. Does it **stop** right?
3. Does it **keep** the invariant true?
4. Does it make **progress** toward termination?
Add elements backward

```c
int s = 0;
int h = 0;
while (h >= 0) {
    s = s + b[h];
    h--;
}
```

INV: \( b \)  
\[
\begin{array}{c|c|c}
0 & ? & s = \text{sum} \\
\end{array}
\]

1. Does it start right?
2. Does it stop right?
3. Does it keep the invariant true?
4. Does it make progress toward termination?
Binary search in sorted b[0..n-1]

Given this precondition and a value v, store a value in h to truthify:

Find invariant by generalizing pre and post
Binary search time \((b[0..n-1] \text{ is sorted})\)

\[
\begin{align*}
h &= -1; \quad t = n; \\
& \quad \text{// invariant: P (below)} \\
& \quad \text{while } (h < t - 1) \\
& \quad \quad \text{int } e = (h + t) / 2; \\
& \quad \quad \text{if } (b[e] \leq v) \quad h = e; \\
& \quad \quad \quad \text{else } \quad t = e; \\
& \quad \} \\
& \quad \text{// } b[0..h] \leq v < b[h+1..] \\
\end{align*}
\]

\[
\begin{array}{l}
\text{b[h+1..t-1] starts out with } n \text{ elements in it.} \\
\text{Each iteration cuts size of } b[h+1..t-1] \text{ in half.} \\
\text{worst-case and expected case time: } \log n
\end{array}
\]
(some) things to know for the prelim

• Can you list the steps in evaluating a new-expression? Can you do them yourself on a piece of paper?
• Can you list the steps in executing a method call? Can you do them yourself on a piece of paper?
• Do you understand exception handling? E.g. What happens after a catch block has been executed?
• Can you write a recursive method or understand a given one?
• Abstract class and interfaces
• ArrayList, interface Comparable
• Loops invariants
private static double m(int x) {
    int y = x;
    try {
        y = 5/x;
        return 5/(x+2);
    } catch (NullPointerException e) {
        System.out.println("null");
    } catch (RuntimeException e) {
        y = 5/(x+1);
    }
    return 1/x;
}
What method calls are legal

Animal an; ... an.m(args);

The ... is computation. stores something in an.

legal ONLY if Java can guarantee that method m exists. How to guarantee?

m must be declared in Animal or inherited. Why?

Someone might write a subclass C of Animal that does not have m declared in it, create an object of C, store it in an. Then method m would not exist.

You know already from lecture 4 on class Object, overriding toString(), and the bottom-up/overriding rule that the overriding method is called.