CSE 8B Today

MORE RECURSION!

NEXT TUESDAY:
Get exam 3 back
(we’ll post grades later today)
We’ll discuss the solutions
1. If we run a recursive method (let’s assume it’s correctly implemented), does it stop calling itself at some point?

A. No, it doesn’t – it continues to call itself forever.

B. No, it doesn’t, that’s why we can’t use such a method.

C. Yes, it stops when it reaches the base case.

D. Yes, it stops when a certain amount of time has passed.

E. Yes, it stops when a certain number of method calls have been done.
2. What happens in Java if we run a recursive method where we forgot (or incorrectly designed) a stopping condition?

A. We can run it and it will terminate normally, it will may just not produce the result we wanted.

B. We can run it, but it will not terminate correctly but causing a StackOverflowError.

C. We can run it, but it will not terminate and keep running forever.

D. We can’t run that method, it will not compile.
3. How many times will each method print “One more!”?
(S.o.p means System.out.println)

A. fooBLUE: 2          fooRED: 2
B. fooBLUE: 2          fooRED: 1
C. fooBLUE: 1          fooRED: 2
D. fooBLUE: 7          fooRED: 7
E. fooBLUE: Infinite   fooRED: 1
4. How many times will this code print the message “One more!”? (S.o.p means System.out.println)

```
public void foo(){
    int x = 3;
    while ( x != 8 ) {
        S.o.p( "One more!" );
        x += 2;
        if (x == 7){
            break;
        }
    }
    S.o.p( "One more!" );
}
```

A. 3
B. 4
C. 7
D. 8
E. Infinite

“LOOPY” review, no recursion here!
public class RecursionDemo
{
    public int g( int x ) { return -1 * x; }
}

//.. In main:
RecursionDemo r = new RecursionDemo();
int result = r.g(-4);

Memory Models (again!) and Recursion

Fill in the memory model for the code at left
public class RecursionDemo {
    public int g( int x ) {
        return -1 * x;
    }
}

//.. In main:
RecursionDemo r = new RecursionDemo();
int result = r.g(-4);
public class RecursionDemo
{
    public int g( int x ) {
        return -1 * x;
    }
    public int f( int x ) {
        return g(x/2);
    }
}

//.. In main:
RecursionDemo r = new RecursionDemo();
r.f(-4);

What does the memory model for this code look like?
Draw the model at the point when g is executing its body.
public class RecursionDemo
{
    public int g( int x ) {
        return -1 * x;
    }
    public int f( int x ) {
        return g(x/2);
    }
}

//.. In main:
RecursionDemo r = new RecursionDemo();
r.f(-4);
“The Stack”

Function data is stored in memory on “the stack”
(objects are stored in a different part of memory called “the heap”)

Every time a function is called, Java makes a new “stack frame”
which stores all of the data for that function (local variables)

(Disclaimer: this is much more complex than this, but close enough for us right now)

For the rest of this lecture (and the whole section on recursion) we will ignore the RecursionDemo object and main, and focus only on the methods the user called explicitly.
public int demo(int x) {
    return x + f(x);
}

public int f(int x) {
    return 11 * g(x) + g(x/2);
}

public int g(int x) {
    return -1 * x;
}

How many stack frames (total) are ever created when we call \texttt{r.demo(-4)}? 

A. 0  
B. 1  
C. 2  
D. 3  
E. 4  

(see next slides)
public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \textit{demo}(-4) ?
What is \texttt{demo(-4)} ?

```java
public int demo(x) {
    return x + f(x);
}
public int f(x) {
    return 11*g(x) + g(x/2);
}
public int g(x) {
    return -1 * x;
}
```
public int demo(int x) {
    return x + f(x);
}
public int f(int x) {
    return 11*g(x) + g(x/2);
}
public int g(int x) {
    return -1 * x;
}

What is `demo(-4)`?
“The Stack” – recursion?

REMEMBER is exactly the same situation... Only that now we call all the time to the same method.
Thinking *sequentially*

**factorial**

\[
5! = 120
\]

\[
5! = 5 \times 4 \times 3 \times 2 \times 1
\]

\[
N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1
\]
Thinking \textit{recursively}

\textbf{factorial}

\begin{align*}
5! & = 120 \\
5! & = 5 \times 4 \times 3 \times 2 \times 1 \\
N! & = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1 \\
N! & = N \times (N-1)! \\
\end{align*}

Recursion == self-reference!
What is wrong with the following?

```java
public int fac(int n) {
    return n * fac(n-1);
}
```

A. A function cannot call itself in Java
B. It will run forever because it is missing a base case
C. It will not compile
D. It will not compute the correct value of factorial because it is missing the recursive step

B. It will run forever because it is missing a base case
public int fac(int n) {
    return fac(n-1);
}
Thinking *recursively*

**factorial**

\[ N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1)! \]  
(And 1! = 1)

Recursive step, do a tiny bit of work, and then assume you have code that solves a smaller version of the same problem.

Base case *(what to do when the problem can’t be simplified any further)*
Complete the factorial method

```java
public int fac( int n ) {
    if ( _________________(1) _____________ )
        ___________ (2)________________
    else
        return n*fac( n-1 );
}
```

What should go in blank (1)?
A. \( n > 1 \)
B. \( n <= 1 \)  \( \text{If we get a 1 or less, we want to stop!} \)
C. \( \text{fac}(n) == 1 \)
D. \( \text{fac}(n) > 1 \)
Complete the factorial method

```java
public int fac( int n ) {
    if ( n <= 1 )
        // (2)
    else
        return n*fac( n-1 );
}
```

What should go in blank (2)?

A. return 1;  
Base case is: 1! = 1
B. return 0;
C. return fac(1);
D. return fac(n-1);
E. return n-1;
How to think *recursively*?

• When writing a recursive method/algorithm you need to do two things:
  
  – **Identify a base case**
    • NO recursive call in the base case
    • A version of the problem so simple that you already know the answer, so just do it!
  
  – **The recursive step**
    • Assume that your function will WORK (magic!) for smaller versions of the input
    • Remember to do a small piece of the work
    • Remember that your new input must be heading toward your base case (UPDATE somehow the parameters that you input in the recursive call)

• You can think about these two steps in any order. Whatever works for you!
How to think recursively?

```java
public static int fac(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fac(n - 1);
    }
}
```

Base case. We know the answer. No recursive call! But you need the if statement to identify that this really IS the base case.

Do a little bit of the work

n-1 is less than n, so we can assume that fac will work on this input. Magically!

Also, n-1 will eventually reach the value 1 (assuming n started positive)
NOTE: All recursive code can be implemented with loops

• Simple: if, else recursion is usually “easy” to turn into a loop

• But some problems are naturally expressed in terms of a recursive structure, which makes writing the method easier and more elegant
// In the class RecursionDemo
public static void recFun( int num ) {
    System.out.println( num );
    num = num - 1
    if ( num > 0 ) {
        recFun( num );
        System.out.println( num );
    }
    // return;
}

NOTE THAT:
1) We print once, before doing -1
2) Another print after the -1 BUT when we “come back” from the recursion.
3) num == 0 reaches our base case!

What does the above code print?

A. 3   B. 3   C. 3   D. 3   E. 3
   3   3   3   3   3
   2   2   2   2   ...
   1   1   1   1   1
   0   0   0   0   0
   ...  ...  ...  ...  ...
Code tracing and recursion

// In the class RecursionDemo
public static void recFun2( int num ) {
    if ( num != 0 ) {
        recFun2( num - 2 );
        System.out.println( num );
    }
}

RecursionDemo.recFun2( 3 )

What does the above code print?

A. Nothing, it never gets to the print statement
B. 3
C. 3
D. 1
E. 3}

We never “come back” from the recursive call... (because we never get a num equal to zero) so we never get the chance to run the println line
Additional questions...

```java
// In the class RecursionDemo
public static void recFun2(int num) {
    if (num != 0) {
        recFun2(num - 2);
        System.out.println(num);
    }
}

RecursionDemo.recFun2(3)
```

What does the above code print?

A. Nothing, it never gets to the print statement  
B. 3 1 1  
C. 3 -1 -3 -5 ...  
D. 1 3 3 3 3 3 3 ...  
E. 3 3 3 3 3 3 3 3 ... 

What small change would make the answer C?

What is the base case here?

num == 0

Can you rewrite the code in `recFun2` to make the answer D (with the same method call)?

Put the print right before the recursive call!