CS1007 lecture #18 notes

thu 15 nov 2001

http://www.cs.columbia.edu/~sklar/cs1007

today:
- news
- recursion
- reading: *ch 11*

news.

— midterm #2 back today, mean=69.33789954

— the FINAL EXAM SCHEDULE:

- **the exam is: Tue Dec 18th 9am-12noon**
- **makeup is: Thu Dec 20th 1pm-4pm**

— there will be a signup for the makeup final after Thanksgiving

— homework #6 is on-line (DUE WED NOV 21, 12NOON)

— there will only be one more homework (#7) after that

Recursion.

- recursion is defining something in terms of itself
- there are many examples in nature
- and in mathematics
- and in computer graphics, e.g., the Koch snowflake (textbook, p.485)

power function.

power is defined recursively:

$$x^y = \begin{cases} \text{if } y == 0, & x^y = 1 \\ \text{if } y == 1, & x^y = x \\ \text{otherwise}, & x^y = x * x^{y-1} \end{cases}$$

here it is in a Java method.

```
public int power ( int x, int y ) {
  if ( y == 0 ) {
    return( 1 );
  }
  else if ( y == 1 ) {
    return( x );
  }
  else {
    return( x * power( x, y-1 ));
  }
} // end of power() method
```

Notice that `power()` calls itself!

You can do this with any method *except main()*

BUT beware of infinite loops!!!

You have to know when and how to stop the recursion — what is the *stopping* condition

let's walk through `power(2,4)`.

	call	x	y	return value
1	power(2,4)	2	4	2 * power(2,3)
2	power(2,3)	2	3	2 * power(2,2)
3	power(2,2)	2	2	2 * power(2,1)
4	power(2,1)	2	1	2

the first is the *original call*

followed by three *recursive calls*

stacks.

- the computer uses a data structure called a *stack* to keep track of what is going on
- think of a *stack* like a stack of plates
- you can only take off the top one
- you can only add more plates to the top
- this corresponds to the two basic *stack operations*:
  - *push* — putting something onto the stack
  - *pop* — taking something off of the stack
- when each recursive call is made, power() is pushed onto the stack
- when each return is made, the corresponding power() is popped off of the stack

another example: factorial.

factorial is defined recursively:

$$N! = \begin{cases} \text{if } N == 1, & N! = 1 \\ \text{otherwise,} & N! = N * (N-1)! \end{cases}$$

(for $N > 0$)

here it is in a Java method.

```
public int factorial ( int N ) {
  if ( N == 1 ) {
    return( 1 );
  }
  else {
    return( N * factorial( N-1 ));
  }
} // end of factorial() method
```


recursive iteration.

You can also use recursion to iterate.

Here's an example:

```
public class ex18a {

  int[] myArray = new int[5];

  public static void main( String[] args ) {
    ex18a ex = new ex18a();
    for ( int i=0; i<5; i++ ) {
      ex.myArray[i] = i + 10;
    }
    ex.printArray( 0 );
  } // end of main() method

  public void printArray( int index ) {
    if ( index < myArray.length ) {
      System.out.print( myArray[index] + " " );
      printArray( index+1 );
    }
    else {
      System.out.println();
    }
  } // end of printArray() method

} // end of ex18a class
```


normal iteration.

in place of what we've done in the past:

```
public class ex18b {

  int[] myArray = new int[5];

  public static void main( String[] args ) {
    ex18b ex = new ex18b();
    for ( int i=0; i<5; i++ ) {
      ex.myArray[i] = i + 10;
    }
    ex.printArray( 0 );
  } // end of main() method

  public void printArray() {
    for ( int index=0; index<myArray.length; index++ ) {
      System.out.print( myArray[index] + " " );
    }
    System.out.println();
  } // end of printArray() method

} // end of ex18b class
```


back to recursive iteration.

in the recursive version, each call is like one iteration inside the for loop in the iterative version

	call	index	output	next call
1	printArray(0)	0	10	printArray(1)
2	printArray(1)	1	11	printArray(2)
3	printArray(2)	2	12	printArray(3)
4	printArray(3)	3	13	printArray(4)
5	printArray(4)	4	14	printArray(5)
6	printArray(5)	5	newline	— —