CS1007 lecture #19 notes

tue 20 nov 2001

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

today:
- news
- finishing up recursion (ch 11)
- inheritance (ch 7)

recursive iteration.

Last class we talked about classic recursion, and went through examples with *power* and *factorial*. You can also use recursion to iterate; this is what we will talk about today.

Here's an example:

```
public class ex19a {

  int[] myArray = new int[5];

  public static void main( String[] args ) {
    ex19a ex = new ex19a();
    for ( int i=0; i<5; i++ ) {
      ex.myArray[i] = i + 10;
    }
    ex.printArray( 0 );
  } // end of main() method

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

} // end of ex19a class
```


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

```
public class ex19b {

  int[] myArray = new int[5];

  public static void main( String[] args ) {
    ex19b ex = new ex19b();
    for ( int i=0; i<5; i++ ) {
      ex.myArray[i] = i + 10;
    }
    ex.printArray( 0 );
  } // end of main() method

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

} // end of ex19b 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	— —

more on recursion.

With recursion, each time the method is invoked, one step is taken towards the resolution of the task the method is meant to complete.

Before each step is executed, the state of the task being completed is somewhere in the middle of being completed.

After each step, the state of the task is one step closer to completion.

In the example above, each time $printArray(i)$ is called, the array is printed from the i-th element to the end of the array.

In the $power(x, y)$ example from last class, each time the method is called, power is computed for each x^y, in terms of the previous x^{y-1}.

In the $factorial(N)$ example from last class, each time the method is called, factorial is computed for each N, in terms of the previous $N - 1$.

The book uses the classic "Towers of Hanoi" example (p477-482). Each time $moveTower()$ is called, one disk is moved from one tower to another. At each point (i.e., at the start of each recursive call), the state of the towers is in the middle of completion, until the final solution is reached.

inheritance.

"Inheritance" is the means by which classes are created out of other classes.

This is a cornerstone of object-oriented programming.

The idea is to create classes that can be re-used from one application to another.

Classes contain "data objects" and "methods". You want to be able to change the data type of the data objects and still be able to use the same methods. You also want to be able to change what the methods do.

inheritance tree.

Think of the most primitive Java class, `Object` as being at the root of the inheritance tree. All other classes are “children” or *subclasses* of that class.

Here is an example of the inheritance tree for Applet:

```
Object
   |
Component
   |
Container
   |
Panel
   |
Applet
```

As you move DOWN the inheritance tree from the root to the leaf, you are *extending* subclasses from parent classes. Parent classes are also called *superclasses*.

As you move UP the inheritance tree from the leaf to the root, you can say that each subclass is a more specific version of its parent. This is known as the *is-a* relationship between a subclass and the parent class that the child extends.

overriding methods.

For homework #5, you had to *extend* the `Applet` class and *override* the `paint()` method.

If you traverse the inheritance tree for `Applet`, you will find that `paint()` is defined in the `Component` class.

In your homework, you wrote a new version of the `paint()` method to draw a soccer ball and two robots. This was called *overriding* the method.

When your homework is executed, your `paint()` method is the one that is invoked, instead of the one in a superclass. The rule is that the version of any method that is invoked is the definition closest to the leaf of the tree.

If you want to refer to the version of the method in a class’s superclass, you use the `super` reference. So in order to invoke the version of `paint()` that is defined in the `Component` class, you call: `super.paint()`

overloading methods.

In addition to changing precisely what a method does, you can also change the arguments to that method.

This is very useful if you are changing the data type of data objects defined in the class.

You can create a new version of a method which has different arguments from the version of the method defined in the class’s superclass.

This is done in the `Block()` constructor in homework #6.