In the last few lectures, we looked at an example of classes whose objects reference each other. This gave us a mechanism to link objects together into a list. We will see other examples later in the course (e.g. trees, graphs).

Let's now turn to *methods*, and consider how methods call/invoke each other. A method is just a sequence of instructions which is encapsulated by a name. When we say that a method is invoked we just mean that the program jumps to the first line of that method, executes the instructions in that method, and then returns to where it was called from.[1]

A brief detour into "how the system works" (see textbook GT Ch. 14.1.1)

When a method is invoked, a chunk of memory is allocated which is called a *stack frame*. Such frames are distinct from the objects and class definitions. Each stack frame stores information that is needed for a particular invocation of a particular method. This information includes:

- the values of the argument(s) that were passed to the method
- values of the local variables of the method
- the address of the instruction from which the method was invoked; the computer needs to return to this instruction after this particular method terminates
- a reference to the object that invoked this method (`'this'`)

You might wonder why we need these stack frames. Can't you just jump from method to method as the program runs? No, that won't work. The problem is that methods can be invoked inside methods (which can be invoked inside methods, which can be invoked inside methods, which can...). Since the program is allowed to jump around from method to method, it needs a way of keeping track of where it came from, so that it can go back again.

[ASIDE: The above discussion was not meant to be complete, rather it was meant to give you a sense of how methods call each other. We will discuss this a bit more in future lectures, but you will have to wait for further courses on "computer systems" before you see a more complete description of how this works.]

Recursion

Let's turn to a particular way that a method can invoke another method. In many programming languages, including Java, it is possible for a method "to invoke itself". This is called *recursion.* We say that a method is *recursive* if it invokes itself. Today we will begin with some simple examples. You have seen recursion in COMP 202, but I will begin with a few examples to refresh you.

[1]Recall that a method can be static, in which case we think of the method's instructions as stored with the class definition, or a method can be non-static, in which case we think of the methods instructions as being stored with a particular instance of the class (an object). Note that instance methods have access to the instance fields of the particular object. Static methods do not have access to the particular fields of an object.

Suppose you wish to ”countdown” the numbers from a given `n` down to 1. The usual way to do it is like this:

```
public static void countdown(int n){  //  assume  n > 1
   while (n > 0) {
       System.out.println(n);
       n-- ;
   }
}
```

Alternatively, you could do it by having the method `countdown` call itself:

```
public static void countdown(int n){  //  assume  n > 1
   if (n > 0) {
       System.out.println(n);
       countdown(n - 1) ;
   }
}
```

Another example (see text p. 134-135) is to compute the product of positive integers from 1 to some given number n, i.e. to compute $n!$ (”n factorial”). The non-recursive method is like this:

```
public static int factorial(int n){  //  assume  n > 1
   int result = 1;
   for (int i = 1;  i < n;  i++)
       result *= i;
   return result;
}
```

and the recursive method is like this:

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

The main difference between this and the countdown example is that the latter returns a value (an integer).

[ASIDE: How does the computer keeps track of the many versions of the method's parameter, in this case the parameter `n`? This parameter changes every time the method is invoked. Recalling the discussion earlier of the 'stack frame', we note that *each time the method* `factorial()` *is invoked, a new stack frame is created and the argument that is passed to this method gets stored in this new stack frame.* Thus, there is no problem that the previous version of n gets “erased”.]

Example with array (not discussed in class)

Let's look at an example where one of the parameters is an array (of primitive variables). Suppose we wish to define a method that prints out a consecutive sequence of elements in the array. We could use a for-loop (not shown). Or, we could do it recursively, as follows:

```
public static void displayArray(float a[ ],  int first,  int last){
   System.out.println(a[first]);
   if (first < last)
      displayArray(a, first+1, last);
}
```

As a slight variation, suppose we wanted to display the array backwards. There are several ways we could do it, for example:

```
public static void displayArrayBackwards(float a[ ],  int first,  int last){
   System.out.println(a[last]);
   if (first < last)
      displayArrayBackwards(a, first, last - 1);
}
```

The next one is a bit more subtle:

```
public static void displayArrayBackwards(float a[ ],  int first,  int last){
   if (first < last)
      displayArrayBackwards(a, first+1, last);
   System.out.println(a[first]);
}
```

For all the above examples, there is no real advantage in writing them recursively. These examples are there just to remind you how recursion works. Let's next turn to an example where recursion *does have a computational advantage.*

Computing x^n

Define a method `power(x,n)`, which computes `x` raised to the power `n`. An iterative (non-recursive) method for doing it is:

```
public static double power(double x, int n){  //  assume  n >= 0
   double val = 1.0;
   for (i = 1;  i <= n;  i++)
       val *= x;
   return val;
}
```

We could write a recursive method, similar to what we did for factorial. But instead, let's look at the much more interesting recursive method:

```
public static double power(double x,  int n){   //  assume n >=0
   if (n == 0)
      return 1.0;
   else{
      double tmp = power(x, n/2);
      if ((n % 2) == 1){
         return tmp*tmp*x;}
      else{
         return tmp*tmp;}
   }
}
```

How does this work? Suppose we are computing $(2.47)^{17}$ so `x` = 2.47 and `n` = 17. Then we would be breaking the problem down as follows:

$$(2.47)^{21} = (2.47)^{10} * (2.47)^{10} * (2.47)$$

So, we would need to evaluate $(2.47)^{10}$ and then perform two multiplications, i.e. square $(2.47)^{10}$ and multiply again by 2.47.

How do we evaluate $(2.47)^{10}$? The recursive method calls `power(2.47, 10)`

$$(2.47)^{10} = (2.47)^5 * (2.47)^5.$$

In turn, $(2.47)^5$ is evaluated recursively by calling `power(2.47, 2)` which in turn calls `power(2.47, 1)` and the recursion terminates there (see the `if (n == 0)` condition.

How many multiplications are performed?

- computing $(2.47)^{21}$ from $(2.47)^{10}$ requires 2 i.e. $(2.47)^{10} * (2.47)^{10} * (2.47)$
- computing $(2.47)^{10}$ from $(2.47)^5$ requires 1, i.e. $(2.47)^{10} = (2.47)^5 * (2.47)^5$
- computing $(2.47)^5$ from $(2.47)^2$ requires 2, namley $(2.47)^2 * (2.47)^2 * (2.47)$
- computing $(2.47)^2$ from $(2.47)^1$ requires 1

So, only 6 multiplications are required, instead of 20 which is what an iterative method would have used. Wow!

[ASIDE: You will be perhaps observe that the process here is similar to that of converting a decimal number into its binary representation. We will see many other examples of such similarities later.]

I finished the lecture by briefly discussing the Tower of Hanoi problem and its recursive solution. See textbook GT, Exercise C-3.11 on p. 151.