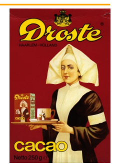
Recursion



Chapter 5 in Rosen
Chapter 11 in Savitch

What does this method do?


```
/**
* precondition n>0
* postcondition ??
*/
private void printStars(int n) {
     if (n == 1) {
          System.out.println("*");
     } else {
          System.out.print("*");
          printStars(n - 1);
     }
}
```

Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.
- **recursive programming**: Writing methods that call themselves
  - directly or indirectly
  - An equally powerful substitute for *iteration* (loops)
  - But sometimes much more suitable for the problem

Definition of recursion

```
recursion: n.
   See recursion.
```

Recursive Acronyms

Dilbert: Wally, would you like to be on my TTP project?
Wally: What does "TTP" stand for?
Dilbert: It's short for **T**he **T**TP **P**roject. I named it myself.
— *Dilbert, May 18, 1994*

GNU — GNU's Not Unix
KDE — KDE Desktop Environment
PHP - PHP: Hypertext Preprocessor
PNG — PNG's Not GIF (officially "Portable Network Graphics")
RPM — RPM Package Manager (originally "Red Hat Package Manager")

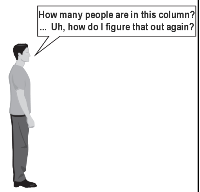
http://search.dilbert.com/comic/Ttp

Why learn recursion?

- A different way of thinking about problems
- Can solve some problems better than iteration
- Leads to elegant, simple, concise code (when used well)
- Some programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

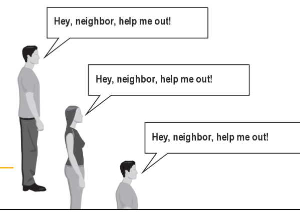
Exercise

- (To a student in the front row)
  How many students are directly behind you?
  - We all have poor vision, and can only see the people right next to us. So you can't just look back and count.
  - But you are allowed to ask questions of the person behind you.
  - How can we solve this problem? (*recursively*)



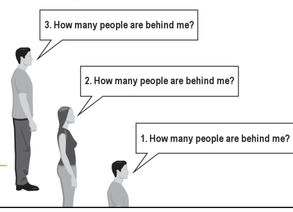
The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help you?



Recursive algorithm

- Number of people behind me:
  - If there is someone behind me, ask him/her how many people are behind him/her.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody behind me, I will answer **0**.



Recursive structures

- A directory has
  - files

  and
  - (sub) directories
- An expression has
  - operators
  - operands, which are
    - variables
    - constants
    - (sub) expressions

Expressions represented by trees

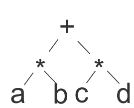
- A tree is
  - a node

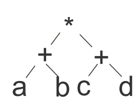
  with
  - zero or more sub trees

examples:

a*b + c*d

(a+b)*(c+d)





Structure of recursion

- Each of these examples has
  - recursive parts (directory, expression, tree)
  - non recursive parts (file, variables, nodes)
- Would we always need non recursive parts?
- Same goes for recursive algorithms.

Cases

- Every recursive algorithm has at least 2 cases:
  - **base case**: A simple instance that can be answered directly.
  - **recursive case**: A more complex instance of the problem that cannot be directly answered, but can instead be described in terms of smaller instances.
  - Can have more than one base or recursive case, but all have at least one of each.
  - A crucial part of recursive programming is identifying these cases.

Base and Recursive Cases: Example

```
public void printStars(int n) {
    if (n == 1) {
        // base case; print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

Recursion Zen

- An even simpler, base case is `n=0`:

```
public void printStars(int n) {
    if (n == 0) {
        // base case; end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

  - **Recursion Zen**: The art of identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Everything recursive can be done non-recursively

```
// Prints a line containing a given number of stars.
// Precondition: n >= 0
public void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

Exercise

- Write a method `reverseLines` that accepts a file `Scanner` prints to `System.out` the lines of the file in reverse order.
  - Write the method recursively and without using loops.
  - Example input: / Expected output:

Example input		Expected output
this	→	no?
is		fun
fun		is
no?		this

  - What are the cases to consider?
    - How can we solve a small part of the problem at a time?
    - What is a file that is very easy to reverse?

Reversal pseudocode

- Reversing the lines of a file:
  - Read a line L from the file.
  - Print the rest of the lines in reverse order.
  - Print the line L.

- If only we had a way to reverse the rest of the lines of the file....

Reversal solution

```
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case
        String line = input.nextLine();
        reverseLines(input);
        System.out.println(line);
    }
}
```

- Where is the base case?

Tracing our algorithm

- **call stack**: The method invocations running at any one time.

```
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {   // false
        ...
    }
}
```

output:

```
no?
fun
is
this
```

input file:

```
this
is
fun
no?
```

Recursive power example

- Write a method that computes x^n.
  $x^n = x * x * x * ... * x$ (n times)
- An iterative solution:

```
public int pow(int x, int n) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product = product * x;
    }
    return product;
}
```


Exercise solution

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public int pow(int x, int n) {
    if (n == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else {
        // recursive case:  x^n = x * x^(n-1)
        return x * pow(x, n-1);
    }
}
```

How recursion works

- Each call sets up a new instance of all the parameters and the local variables
- When the method completes, control returns to the method that invoked it (which might be another invocation of the same method)

```
pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 1
          = 64
```


Infinite recursion

- A method with a missing or badly written base case can causes **infinite recursion**

```
public int pow(int x, int y) {
    return x * pow(x, y - 1);  // Oops! No base case
}

pow(4, 3) = 4 * pow(4, 2)
          = 4 * 4 * pow(4, 1)
          = 4 * 4 * 4 * pow(4, 0)
          = 4 * 4 * 4 * 4 * pow(4, -1)
          = 4 * 4 * 4 * 4 * 4 * pow(4, -2)
          = ... crashes: Stack Overflow Error!
```

An optimization

- Notice the following mathematical property:

  $$3^{12} = (3^2)^6 = (9)^6 = (81)^3 = 81 * (81)^2$$

  - How does this "trick" work?
  - Do you recognize it?
  - How can we incorporate this optimization into our `pow` method?
  - What is the benefit of this trick?
  - Go write it.

Exercise solution 2

```java
// Returns base ^ exponent.
// Precondition: exponent >= 0
public int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else if (exponent % 2 == 0) {
        // recursive case 1:  x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {
        // recursive case 2:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

Activation records

- **Activation record**: memory that Java allocates to store information about each running method
  - return point ("RP"), argument values, local variables
  - Java stacks up the records as methods are called; a method's activation record exists until it returns
  - Eclipse debug draws the act. records and helps us *trace* the behavior of a recursive method

```
 ______________________________
| x  = [ 4 ]         n = [ 0 ] | pow(4, 0)
|_RP = [pow(4,1)]______________|
| x  = [ 4 ]         n = [ 1 ] | pow(4, 1)
|_RP = [pow(4,2)]______________|
| x  = [ 4 ]         n = [ 2 ] | pow(4, 2)
|_RP = [pow(4,3)]______________|
| x  = [ 4 ]         n = [ 3 ] | pow(4, 3)
|_RP = [main]__________________|
|______________________________| main
```

27