Credits: (1) Marty Stepp and Hélène Martin
Building Java Programs
(2) Chris Kiekintveld
Elementary Data Structures and Algorithms

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.

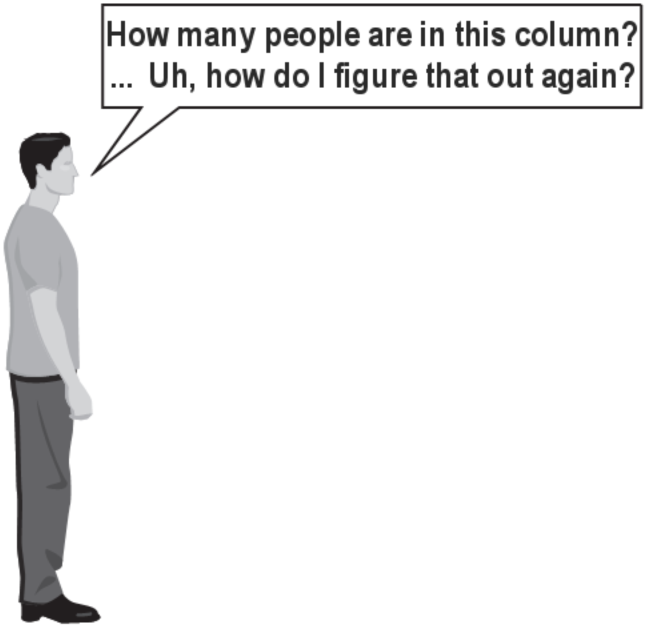
- **recursive programming**: Writing methods that call themselves to solve problems recursively.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems

Why learn recursion?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and CommonLisp) use recursion exclusively (no loops)

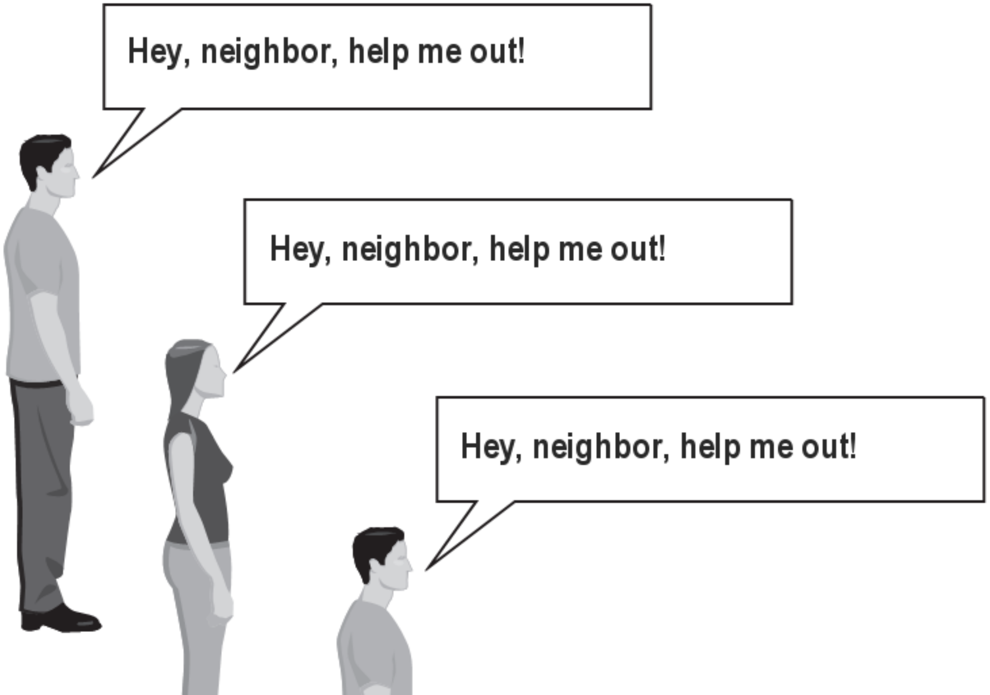
Exercise

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


How many people are in this column?
... Uh, how do I figure that out again?

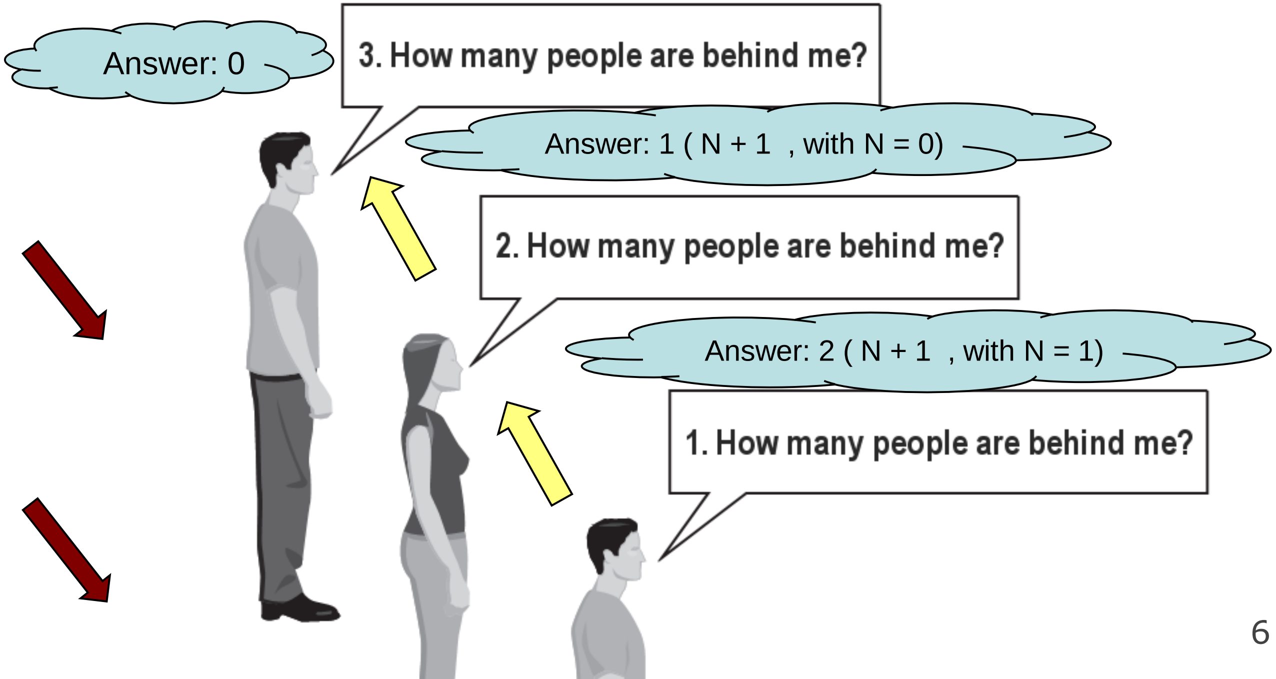
The idea

- 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 me?


Hey, neighbor, help me out!
Hey, neighbor, help me out!
Hey, neighbor, help me out!

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**.



Answer: 0

3. How many people are behind me?

Answer: 1 (N + 1 , with N = 0)

2. How many people are behind me?

Answer: 2 (N + 1 , with N = 1)

1. How many people are behind me?

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case**: A simple occurrence that can be answered directly.

    VERY IMPORTANT

    Stops the recursion (prevents infinite loops)

    Solved directly to return a value without calling the same method again

  - **recursive case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

  - Some recursive algorithms 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.

Can you identify the base case and recursion case ?

```
 int  binSearch(int A[ ], int lower, int upper, int X)
{
     // check case for missing X
     if (lower > upper)                    Base Case
        return -1;

    // check if X is at the middle
     int middle = (lower + upper)/2;
      if (A[middle] == X)                  Base Case
        return middle;

                                           Recursion Case
     if (A[middle] < X)
        return binSearch(A, middle+1, upper, X);
     else                                  Recursion Case
        return binSearch(A, lower, middle-1, X);
}
```

Recursion in Java

- Consider the following method to print a line of `*` characters:

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

- Write a recursive version of this method (that calls itself).
  - Solve the problem without using any loops.
  - Hint: Your solution should print just one star at a time.

A basic case

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

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

Handling more cases

- Handling additional cases, with no loops (in a bad way):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...
}
```

Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1);    // prints "*"
    } else if (n == 3) {
        System.out.print("*");
        printStars(2);    // prints "**"
    } else if (n == 4) {
        System.out.print("*");
        printStars(3);    // prints "***"
    } else ...
}
```

Using recursion properly

- Condensing the recursive cases into a single case:

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

What is happened if we switch the print(*); ?

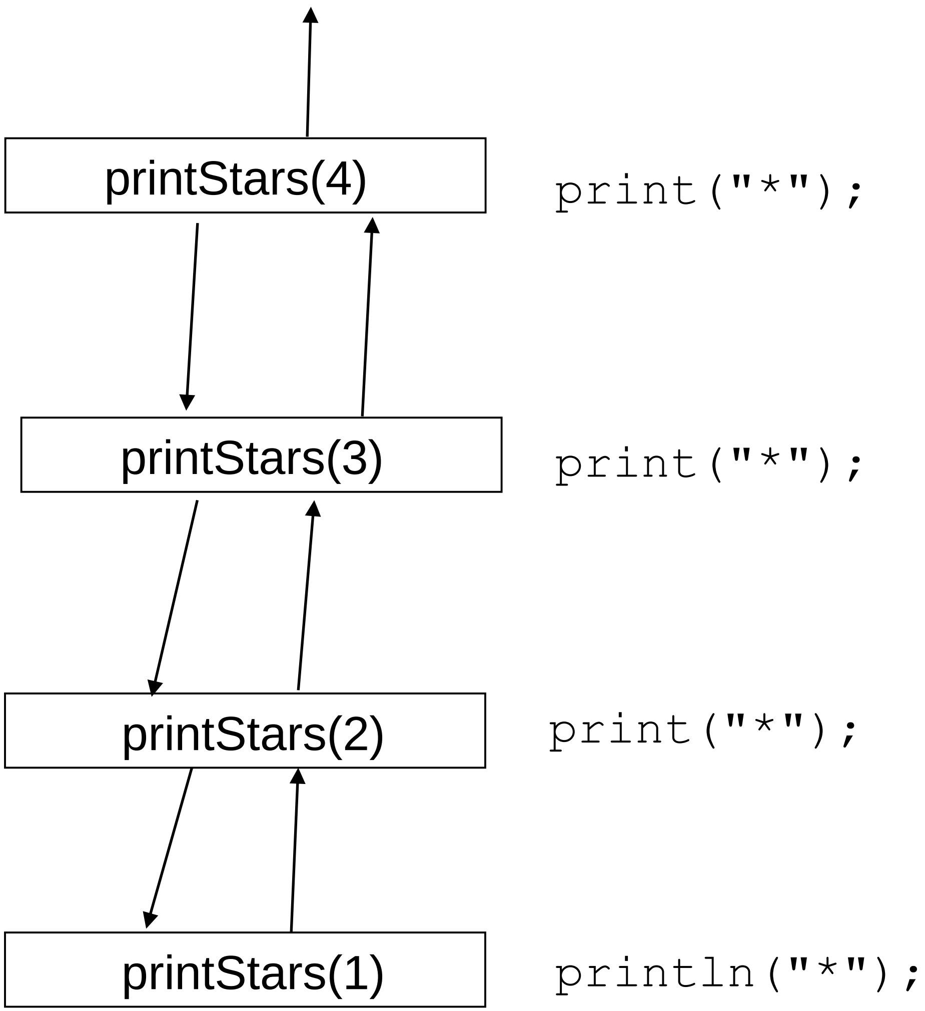
- Condensing the recursive cases into a single case:

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

```
«GRASP exec: java Test

  *
  ***
 ----jGRASP: operation complete.
```

Explain: What is happened if we switch the print(*); ?



printStars(4)

```
print("*");
```

printStars(3)

```
print("*");
```

printStars(2)

```
print("*");
```

printStars(1)

```
println("*");
```

"Recursion Zen"

- The real, even simpler, base case is an `n` of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just 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 properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

A recursive trace

□ As always, go line by line

□ Recursive methods may have many copies

□ Every method call creates a new copy and transfers flow of control to the new copy

□ Each copy has its own:

- ✓ code
- ✓ parameters
- ✓ local variables

A recursive trace

After completing a recursive call:

□ Control goes back to the calling environment

□ Recursive call must execute completely before control goes back to previous call

□ Execution in previous call begins from point immediately following recursive call

Recursive tracing 1

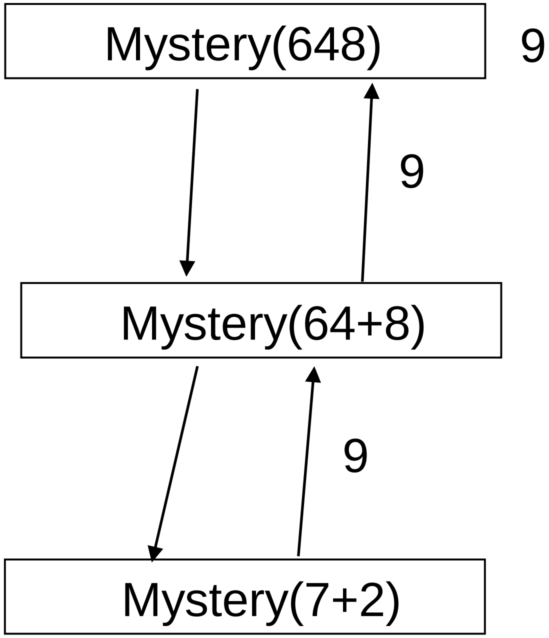
- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

  - What is the result of the following call?

    ```
    mystery(648)
    ```

Recursive tracing 1 (Recursion Tree)



A recursive trace 1

```
mystery(648):
  ▪ int a = 648 / 10;          // 64
  ▪ int b = 648 % 10;          //  8
  ▪ return mystery(a + b);     // mystery(72)

    mystery(72):
    ▪ int a = 72 / 10;         // 7
    ▪ int b = 72 % 10;         // 2
    ▪ return mystery(a + b);   // mystery(9)

      mystery(9):
      ▪ return 9;
```

Recursive tracing 2

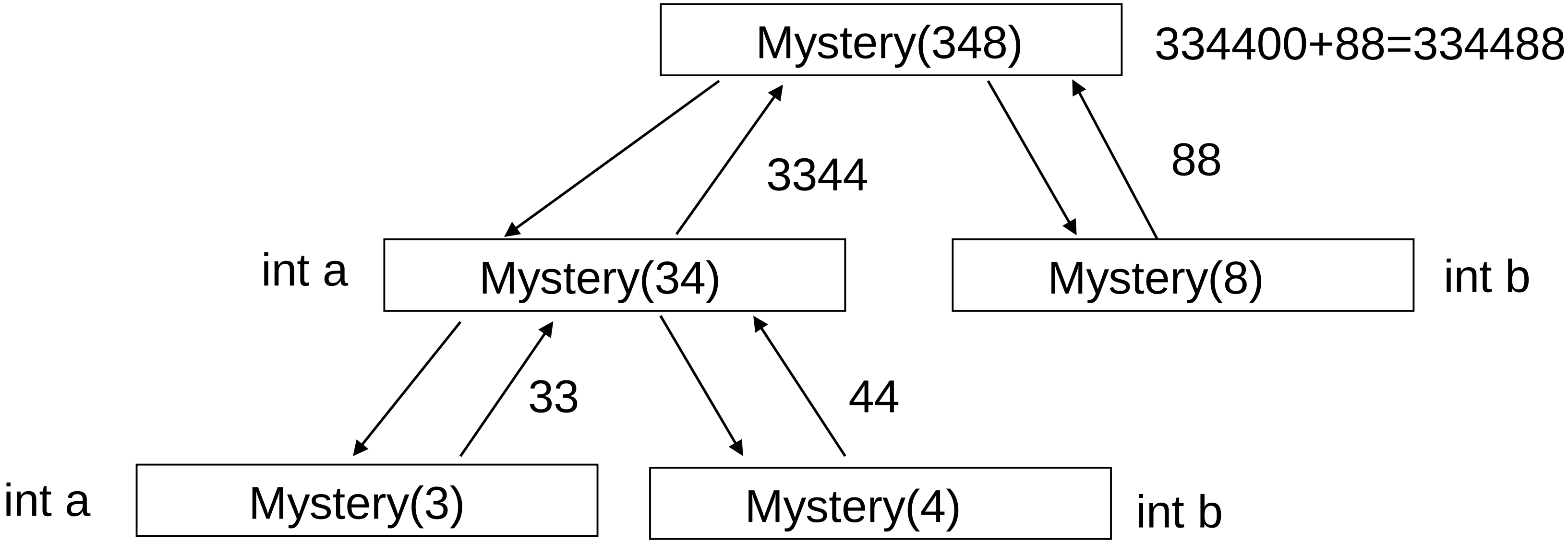
- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?

    ```
    mystery(348)
    ```

Recursive tracing 2 (Recursion Tree)



A recursive trace 2

```
mystery(348)
  ▪ int a = mystery(34);
      • int a = mystery(3);
          return (10 * 3) + 3;       // 33
      • int b = mystery(4);
          return (10 * 4) + 4;       // 44
      • return (100 * 33) + 44;      // 3344

  ▪ int b = mystery(8);
      return (10 * 8) + 8;           // 88

  – return (100 * 3344) + 88;        // 334488
```

– What is this method really doing?

Designing Recursive Algorithms

- General strategy: “Divide and Conquer”

- Questions to ask yourself
  - ❑ How can we reduce the problem to smaller version of the same problem?
  - ❑ How does each call make the problem smaller?
  - ❑ What is the base case?
  - ❑ Will you always reach the base case?

Exercise

- Write a recursive method `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.
  - `isPalindrome("madam")` → `true`
  - `isPalindrome("racecar")` → `true`
  - `isPalindrome("step on no pets")` → `true`
  - `isPalindrome("able was I ere I saw elba")` → `true`
  - `isPalindrome("Java")` → `false`
  - `isPalindrome("rotater")` → `false`
  - `isPalindrome("byebye")` → `false`
  - `isPalindrome("notion")` → `false`

Exercise solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true;   // base case
    } else {
        char first = s.charAt(0);
        char last  = s.charAt(s.length() - 1);
        if (first != last) {
            return false;
        }              // recursive case
        String middle = s.substring(1, s.length() - 1);
        return isPalindrome(middle);
    }
}
```

Exercise solution 2

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true;   // base case
    } else {
        return s.charAt(0) == s.charAt(s.length() - 1)
            && isPalindrome(s.substring(1, s.length() - 1));
    }
}
```

Exercise

- Write a recursive method `reverseLines` that accepts a file `Scanner` and prints the lines of the file in reverse order.
  - Example input file:

    ```
    Roses are red,
    Violets are blue.
    All my base
    Are belong to you.
    ```

    Expected console output:

    ```
    Are belong to you.
    All my base
    Violets are blue.
    Roses are red,
    ```

  - 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 static 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.

```
    reverseLines(new Scanner("poem.txt"));
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "Roses are red,"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "Violets are blue."
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "All my base"
```

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine();   // "Are belong to you."
```

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

```
Roses are red,
Violets are blue.
All my base
Are belong to you.
```

```
Are belong to you.
All my base
Violets are blue.
Roses are red,
```