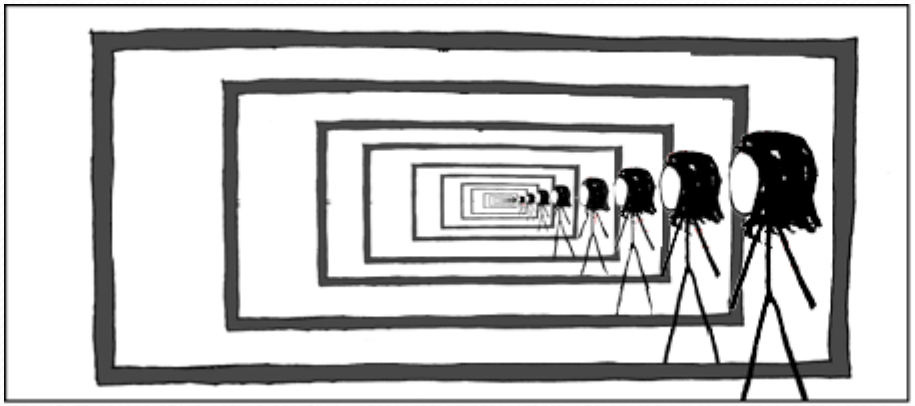
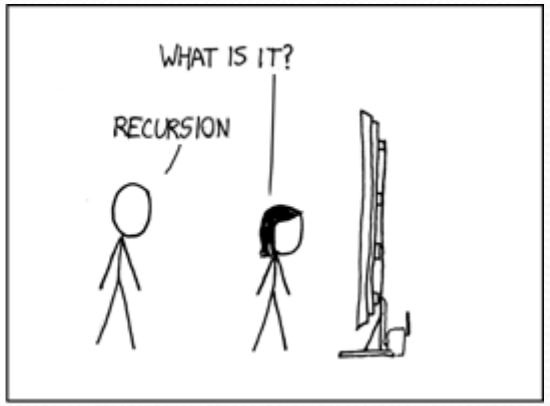
# CS 142

**Lecture 13:** `static`; file I/O; 2D arrays






Thanks to Marty Stepp and Stuart Reges for parts of these slides.

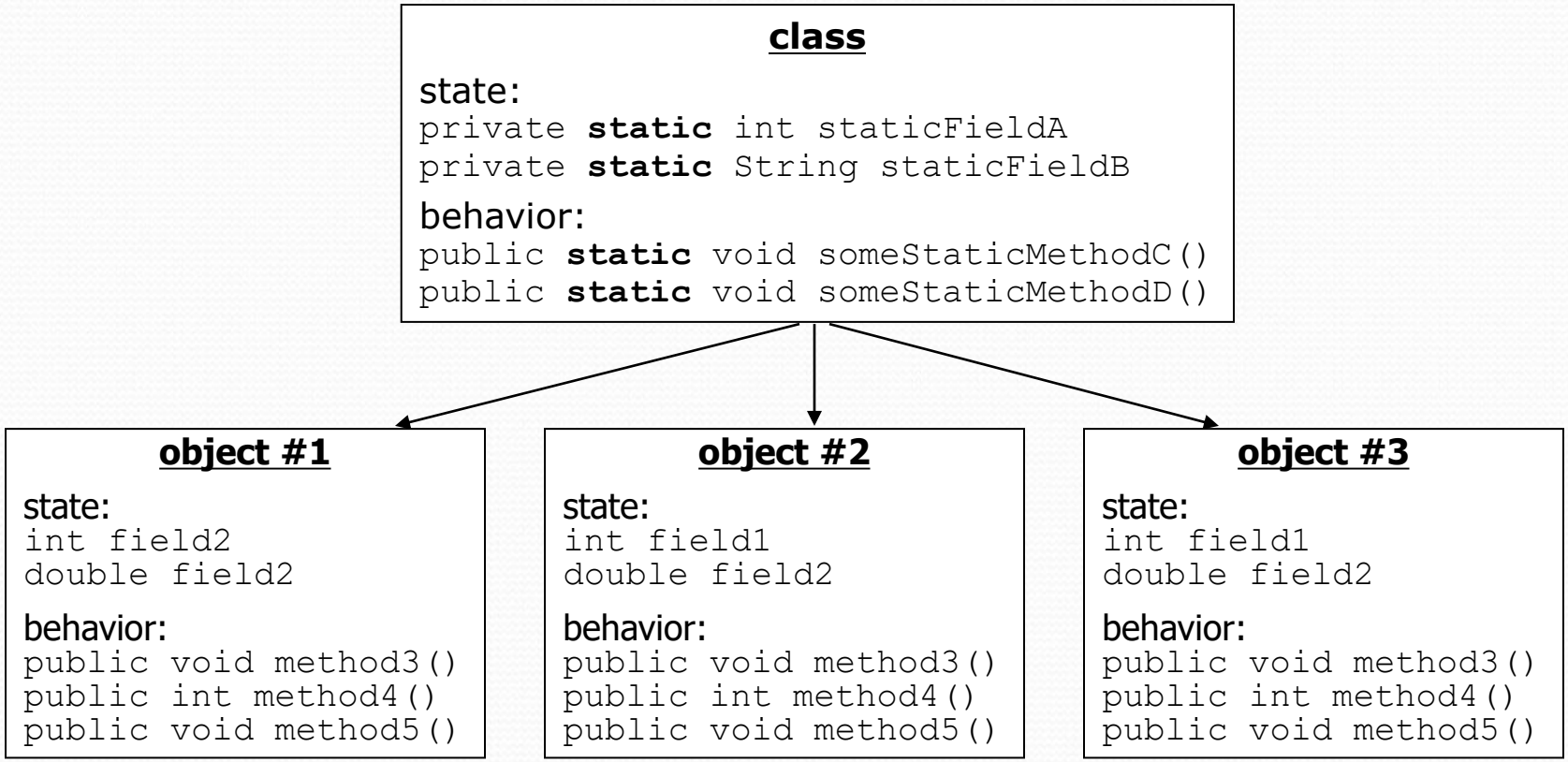
# Exercise: BankAccount

How can we make sure that every bank account gets a unique id?

```
public class BankAccount {
    private String name;
    private int id;

    public BankAccount(int accountNumber) {
        id = accountNumber;  // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

# Static members

- **static**: Part of a class, rather than part of an object.
  - Object classes can have static methods *and fields*.
  - Not copied into each object; shared by all objects of that class.



# Static fields

```
private static type name;
```

or,

```
private static type name = value;
```

- Example:

```
private static int theAnswer = 42;
```

- **static field**: Stored in the class instead of each object.
  - A "shared" global field that all objects can access and modify.
  - Like a class constant, except that its value can be changed.

# Accessing static fields

- From inside the class where the field was declared:

```
fieldName                          // get the value
fieldName = value;                 // set the value
```

- From another class (if the field is `public`):

```
ClassName.fieldName                // get the value
ClassName.fieldName = value;       // set the value
```

  - generally static fields are not `public` unless they are `final`

- Exercise: Write the working version of `Giraffe`.

# BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

# Static methods

```
// the same syntax you've already used for methods
public static type name(parameters) {
    statements;
}
```

- **static method**: Stored in a class, not in an object.
  - Shared by all objects of the class, not replicated.
  - Does not have any *implicit parameter*, `this`; therefore, cannot access any particular object's fields.

- Exercise: Make it so that clients can find out how many total `BankAccount` objects have ever been created.

# BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // clients can call this to find out # accounts created
    public static int getNumAccounts() {
        return objectCount;
    }

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }

    ...
    public int getID() {    // return this account's id
        return id;
    }
}
```

# Creating an array of arrays

`int[10][8] data` creates an array of length 10 that stores arrays of length 8

```
int[10][8] data;

for(int i = 0; i < data.length; i++) {
    for(int j = 0; j < data[i].length; j++) {
        data[i][j] = i * 10 + j;
    }
}

for(int i = 0; i <  data.length; i++) {
    for(int j = 0; j <  data[i].length; j++) {
        System.out.print(data[i][j] + " ");
    }
     System.out.println();
}
```

**Output:**

```
0 1 2 3 4 5 6 7
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
60 61 62 63 64 65 66 67
70 71 72 73 74 75 76 77
80 81 82 83 84 85 86 87
90 91 92 93 94 95 96 97
```

# Array of arrays mystery

```
void mystery(int[][] data, int[] result, int pos, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            result[j] = data[i + pos][j + pos];
        }
    }
}
```

Suppose that a variable called grid has been declared as follows:

```
int[6][6] grid = {{8, 2, 7, 8, 2, 1}, {1, 5, 1, 7, 4, 7},
                  {5, 9, 6, 7, 3, 2}, {7, 8, 7, 7, 7, 9},
                  {4, 2, 6, 9, 2, 3}, {2, 2, 8, 1, 1, 3}}
```

which means it will store the following 6-by-6 grid of values:

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2 | 7 | 8 | 2 | 1 |
| 1 | 5 | 1 | 7 | 4 | 7 |
| 5 | 9 | 6 | 7 | 3 | 2 |
| 7 | 8 | 7 | 7 | 7 | 9 |
| 4 | 2 | 6 | 9 | 2 | 3 |
| 2 | 2 | 8 | 1 | 1 | 3 |

For each call at right, indicate what value is returned. If the function call results in an error, write error instead.

| Function Call | Contents of Array Returned |
|---|---|
| `mystery(grid, 2, 2)` | _____ |
| `mystery(grid, 0, 2)` | _____ |
| `mystery(grid, 3, 3)` | _____ |

# Exercise

Write a function called `max_row` that takes a list of lists as a parameter and returns the index of the row that contains the maximum value.

To test this create an array as follows:

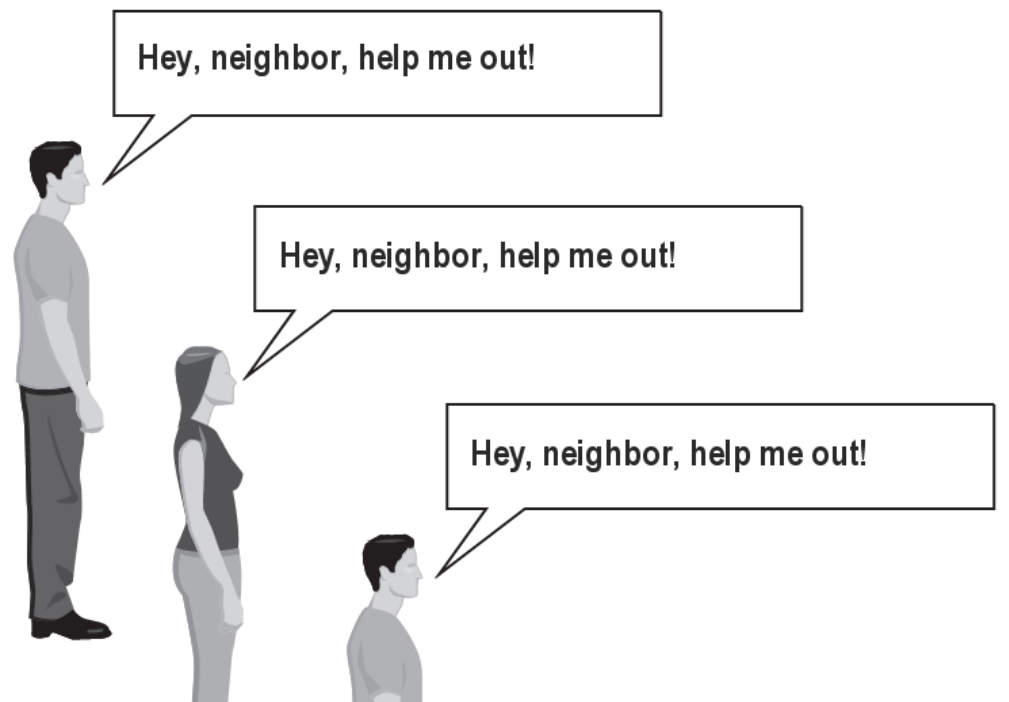
```
int[][] arr =  {0, 1, 2, 3},
               {6, 4, 5, 7},
               {1, 7, 6, 8},
               {5, 2, 6, 7}
              };
```

# 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* )

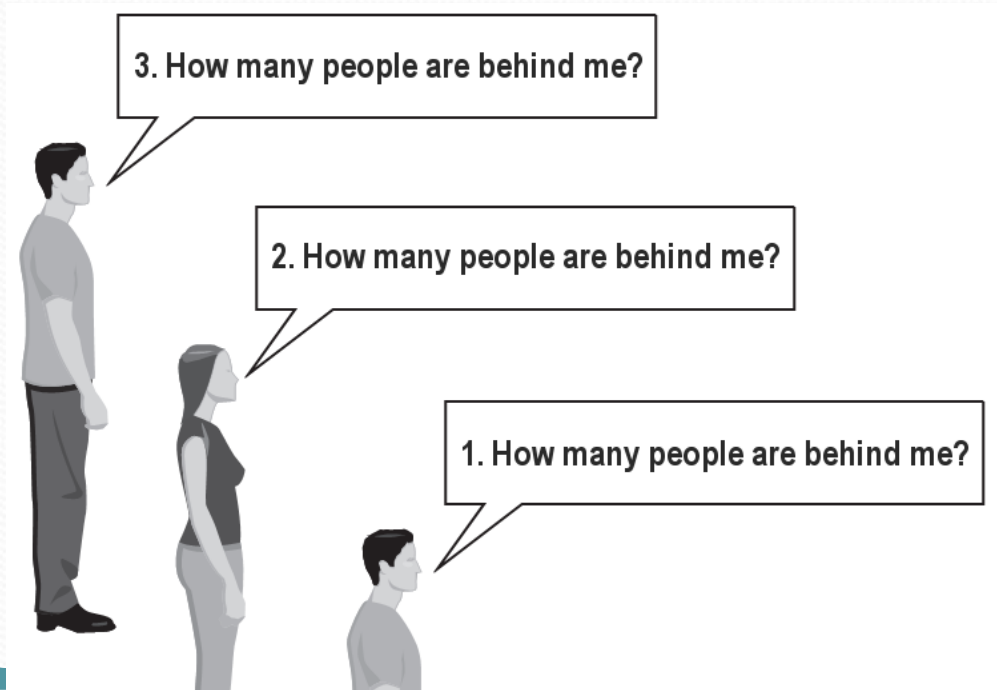
# The idea

- 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?



# 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**.



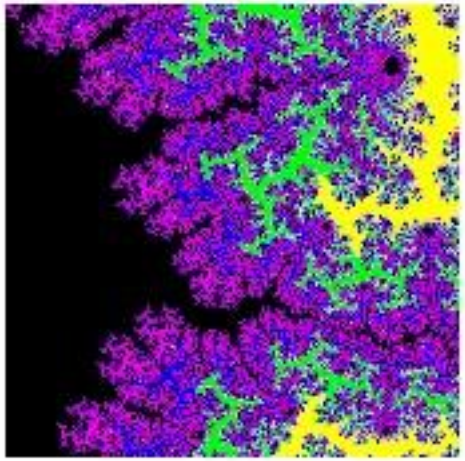
# Recursion

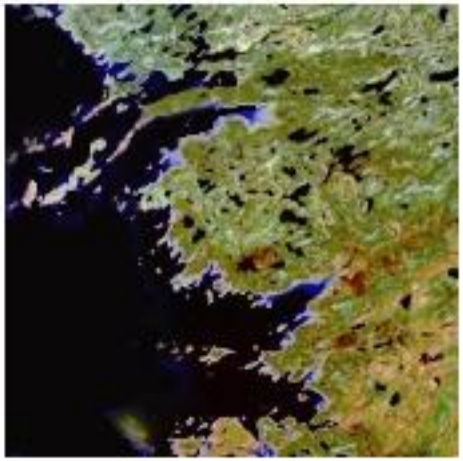
- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- **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


ALOACEAE
ALOE
POLYPHYLLA
SPIRAL ALOE
S. AFRICA

a) 

b) 

a) Part of the Mandelbrot set.

b) Part of the North American coastline near Hudson Bay.

# Why learn recursion?

- "Cultural experience" – think differently about problems
- Solves some problems more naturally than iteration
- Can lead to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- A key component of many of our assignments in CSE 143

# Getting down stairs

- Need to know two things:
  - Getting down one stair
  - Recognizing the bottom

- Most code will look like:

```
if (simplest case) {
    compute and return solution
} else {
    divide into similar subproblem(s)
    solve each subproblem recursively
    assemble the overall solution
}
```

# Recursion and cases

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

# Another recursive task

- How can we remove exactly half of the M&M's in a large bowl, without dumping them all out or being able to count them?
  - What if multiple people help out with solving the problem? Can each person do a small part of the work?
  - What is a number of M&M's that it is easy to double, even if you can't count?
    - (What is a "base case"?)

# 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.