Recursion

MSCI 240: Algorithms & Data Structures

lecture summary

introduction to recursion

recursion and cases

recursive traces and stack traces

public/private pairs

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity	chapter 13	1.4
Searching		pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
Binary Search Trees	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

recursion is the act of defining an object or solving a problem in terms of itself

The Little Schemer
by D. P. Friedman & M. Felleisen, p. xi

recursive programming: writing methods that call themselves to solve problems recursively

- alternative to iteration (loops) – equally powerful
- well-suited to certain types of problems

why use/learn recursion?

"cultural experience"—different way of thinking

"better" than iteration for some problems

leads to elegant, simplistic, short code

many languages use only recursion (no loops)
    "functional" languages (e.g., Scheme, ML, & Haskell)

basic principle: break a big (hard) problem down into smaller occurrences of the same problem

example: count people in a line (students in a class)

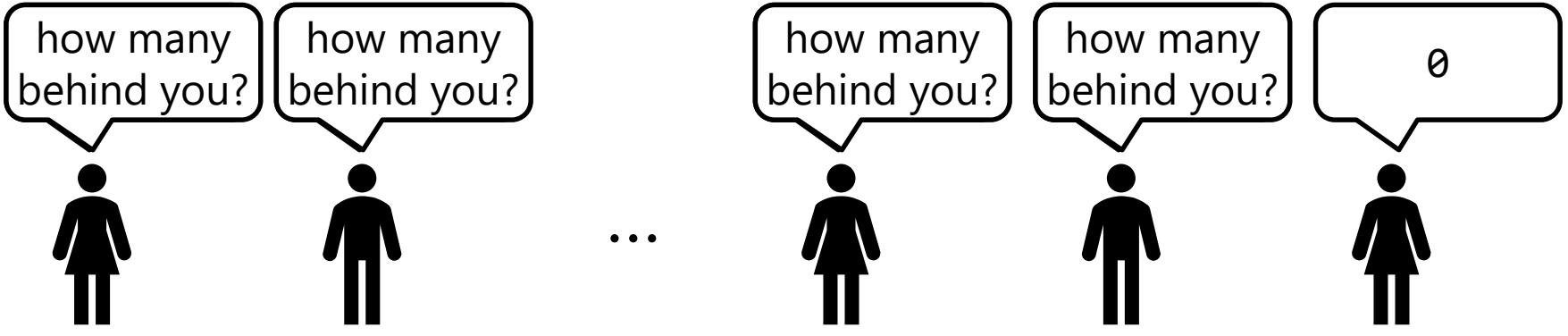
iterative solution

one person counts each person

recursive solution

ask person behind: “how many behind you?”

last person knows (0)



algorithm “how many behind me”:

```
if someone behind me {
   ask how many people behind that person
   when they respond with n, I answer n+1
}
else if no one behind me {
   I answer 0
}
```

```
int howManyBehind(Person me) {
   if someone behind me {

      return howManyBehind(next person) + 1
   }
   if no one behind me {
      return 0
   }
}
```

recursion and cases

every recursive algorithm involves at least two cases:

- base case: a simple occurrence that can be answered directly
- recursive case: a more complex occurrence of the problem that can't 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 cases

rules for writing recursive methods

1. when recurring, two questions/cases:
   a) is this the base case?
      // do something and return

   b) else (is this the recursive case?)
      // enter recursion

2. always change at least one argument in 1.b)
   change must get you closer to termination

   changing argument must be checked in base case

example: factorial (whiteboard)

example: print stars

rewrite the following method using recursion (no loops):

```
// 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
}
```

base case

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

multiple cases (bad!)

```
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 {    //...
}
```

multiple cases (a little 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);
    } else if (n == 3) {
        System.out.print("*");
        printStars(2);
    } else {    //...
}
```

proper recursion

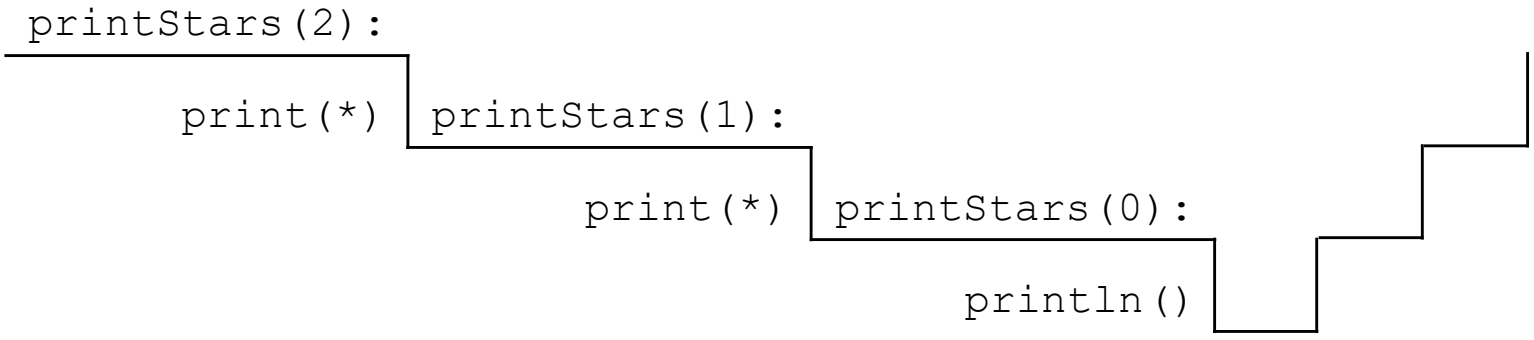
```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n-1);
    }
}
```

recursion "Zen"

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

trace what happens when you call `printStars(2);`

```
printStars(2):
    if (n == 0) {
        // base case; just end the line of input
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(1):
            if (n == 0) {
                // base case; just end the line of input
                System.out.println();
            } else {
                // recursive case; print one more star
                System.out.print("*");
                printStars(0):
                    if (n == 0) {
                        // base case; just end the line of input
                        System.out.println();
                    } else {
                        // recursive case; print one more star
                        System.out.print("*");
                        printStars(n-1);
                    }
            }
    }
```


printStars(2):
print(*)
printStars(1):
print(*)
printStars(0):
println()

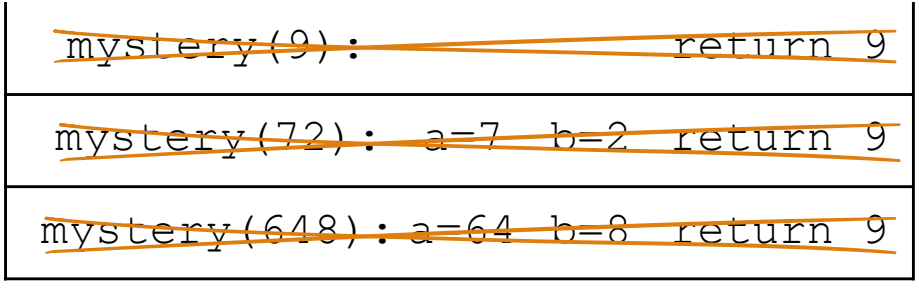
recall: `StackTraceExample` – used a stack to "remember" what method we were in

recursive tracing helps understand what the recursive code is doing

```
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:

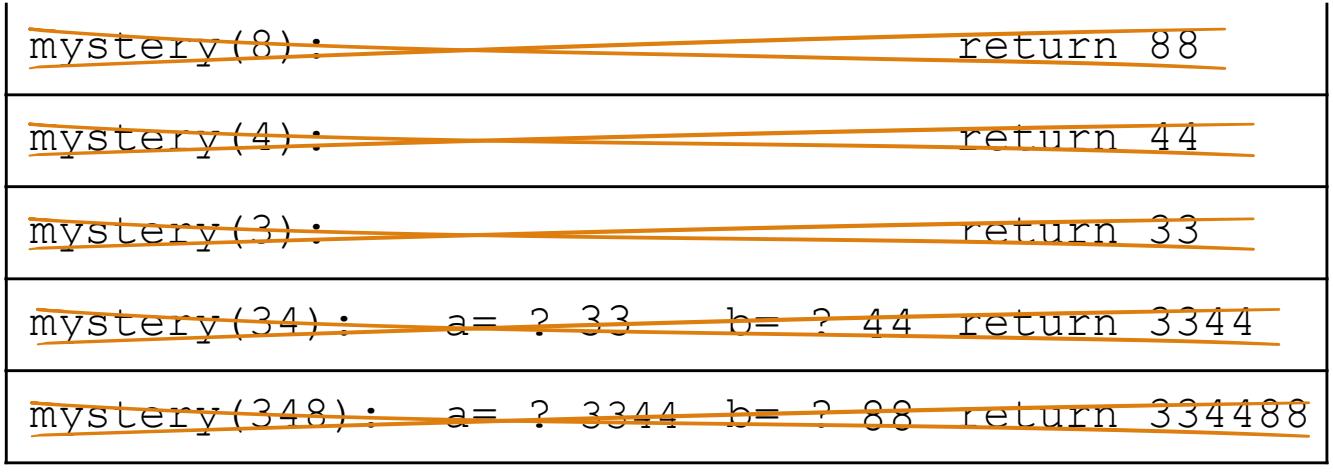
```
mystery(648);
```



```
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:

```
mystery(348);
```



public/private pairs

write a method, `crawl`, that accepts a `File` parameter and prints information about that file

- if the `File` object represents a normal file, just print its name
- if the `File` object represents a directory, print its name and information about every file/directory inside it, indented

```
MSCI121
    Slides
        L01-introduction.pptx
        L02-abstraction-classes.pptx
        L06-encapsulation.pptx
    Homework
        MSCI-240-Fall-2018-HW1_v1.pdf
        MSCI-240-Fall-2018-HW2_v1.pdf
```

recursive data: a directory can contain other directories

A `File` object (from the `java.io` package) represents a file or directory on the disk.

Constructor/method	Description
`File(`**`String`**`)`	creates `File` object representing file with given name
`canRead()`	returns whether file is able to be read
`delete()`	removes file from disk
`exists()`	whether this file exists on disk
`getName()`	returns file's name
`isDirectory()`	returns whether this object represents a directory
`length()`	returns number of bytes in file
`listFiles()`	returns a `File[]` representing files in this directory
`renameTo(`**`File`**`)`	changes name of file

write a method, `crawl`, that accepts a `File` parameter and prints information about that file

- if the `File` object represents a normal file, just print its name
- if the `File` object represents a directory, print its name and information about every file/directory inside it, indented

```
MSCI121
    Slides
        L01-introduction.pptx
        L02-abstraction-classes.pptx
        L06-encapsulation.pptx
    Homework
        MSCI-240-Fall-2018-HW1_v1.pdf
        MSCI-240-Fall-2018-HW2_v1.pdf
```

recursive data: a directory can contain other directories

```java
// Prints information about this file,
// and (if it is a directory) any files inside it.
public static void crawl(File f) {
    crawl(f, ""); // call private recursive helper
}

// Recursive helper to implement crawl/indent behavior.
private static void crawl(File f, String indent) {
    System.out.println(indent + f.getName());
    if (f.isDirectory()) {
        // recursive case; print contained files/dirs
        for (File subFile : f.listFiles()) {
            crawl(subFile, indent + "\t");
        }
    }
}
```

public/private pairs

we can't vary the indentation without an extra parameter:

```
public static void crawl(File f, String indent) {
```

often the parameters we need for our recursion do not match those the client will want to pass

in these cases, we instead write a pair of methods:

1. a **public**, non-recursive one with parameters the client wants
2. a **private**, recursive one with parameters we really need

recursion summary

alternative to iteration (loops)

involves methods that call themselves

two cases to consider:

- base case(s)
- recursive case(s)

can trace a recursive method

sometimes useful to have public/private pair for recursion

next:

sorting