Lecture 19: Recursion

Building Java Programs: A Back to Basics Approach

by Stuart Reges and Marty Stepp

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 Haskell) use recursion exclusively (no loops)

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.

Example

You are lined up in front of your favorite store for Black Friday deals. The line is long and wraps around the building so that you cannot see the front of the line. How do you figure out your position without getting out of line?

Answer: Ask the person in front of you.

Base case: If a customer is at the front of the line and someone asks him for his position, he'll "return" 1.

Recursive case: If a customer is at position n and someone asks him for his position, he'll ask the person in front of him.

Note: The recursive case reduces an n problem to an n-1 problem.

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.

"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.

Exercise

- Write a recursive method `pow` accepts an integer base and exponent and returns the base raised to that exponent.
  - Example: `pow(3, 4)` returns 81
  - Solve the problem recursively and without using loops.

pow solution

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

Recursive tracing

- 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)
```

A recursive trace

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)
```

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?

Recursive trace 3

```
int mystery(int n){
    if (n == 1 || n == 2)
        return 2*n;
    else
        return mystery(n-1) - mystery(n-2);
}
```

What is the result of the following call?

```
mystery(4)
```

Answer: -2

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 {
        String first = s.substring(0,1);
        String last  = s.substring(s.length() - 1);
        if (!first.equals(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.substring(0,1).equals(s.substring(s.length()-1))
            && isPalindrome(s.substring(1, s.length() - 1));
     }
}
```

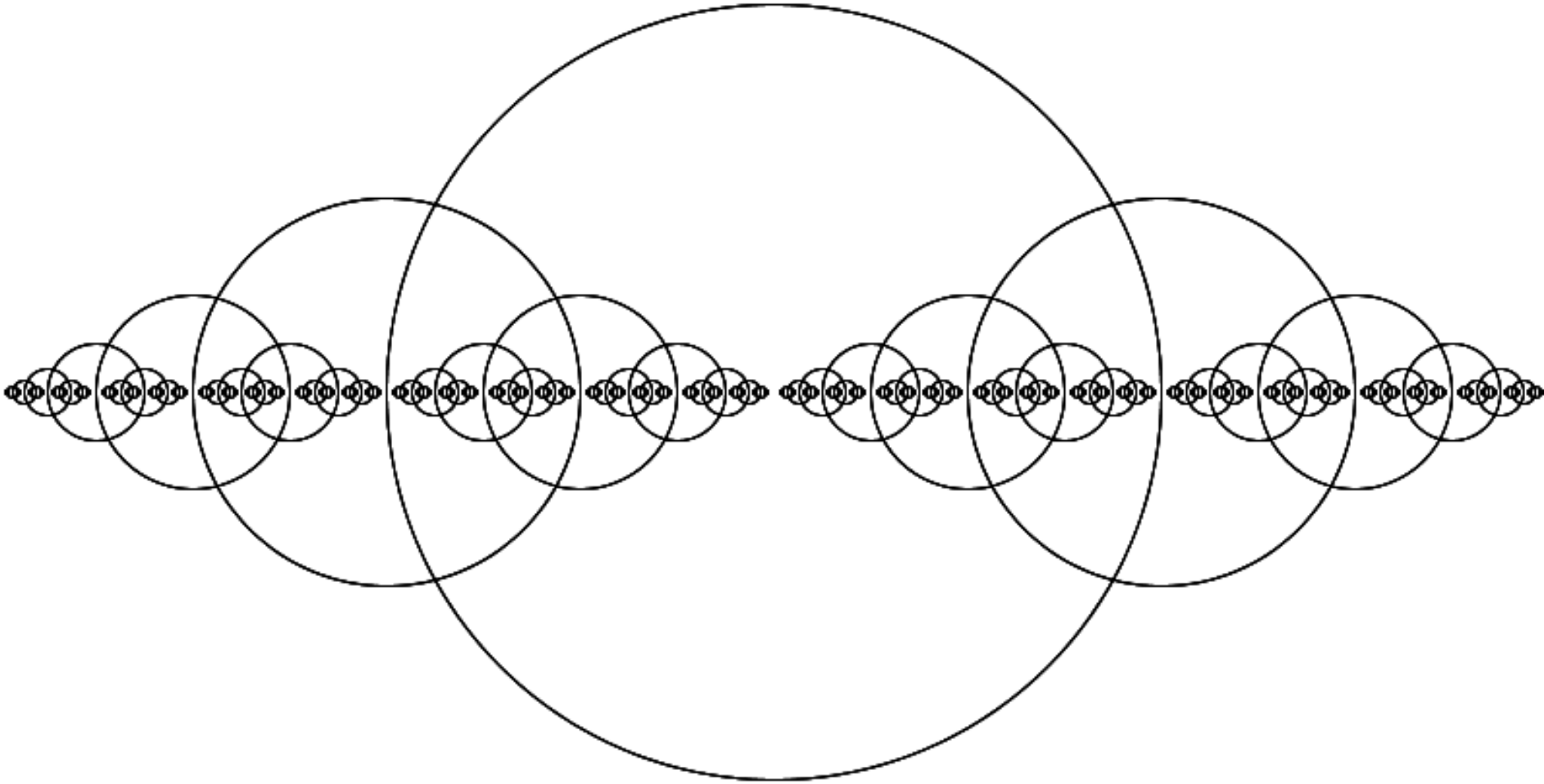
Lab 1: Fractal Circles

Use Processing to write the recursive method to print out a recursive pattern of circles of decreasing radii.

```
void circle(int x, int radius, int depth)
{…}
```

The call circle(width/2, width/4, 10) should produce the image on the following slide. Use noFill() before drawing the circle to make it transparent.

Lab 1



Lab 2: Sierpinski Triangle

Use Processing to write the recursive method to the Sierpinski triangle.

```
void fractalTriangle(int x1, int y1, int x2, int y2,
  int x3, int y3, int n)

{…}
```

Lab 2