Lecture 14: Strings and Recursion

AP Computer Science Principles

Strings

Strings

- **string**: An object storing a sequence of text characters.
  - String is not a primitive type. String is an object type.

```
String name = "text";
String name2 = new String(“text”);
String name3 = expression;
```

  - Examples:

```
String name = ”Mike";
String name2 = new String(“John”);
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
// (3, 5)
```

Indexes

- Characters of a string are numbered with 0-based *indexes*:

```
String name = "J. Smith";
```

index	0	1	2	3	4	5	6	7
character	J	.		S	m	i	t	h

  - First character's index : 0
  - Last character's index : 1 less than the string's length

  - The individual characters are values of type `char`

`String` methods

Method name	Description
`indexOf(`**`str`**`)`	Returns index where the start of the given string appears in this string (-1 if not found)
`length()`	Returns number of characters in this string

`substring(`**`index1`**`, `**`index2`**`)`
or
`substring(`**`index1`**`)`
	Returns the characters in this string from *index1* (inclusive) to *index2* (exclusive);
	if *index2* is omitted, grabs till end of string

`toLowerCase()`	Returns a new string with all lowercase letters
`toUpperCase()`	Returns a new string with all uppercase letters

- These methods are called using the dot notation

String method examples

```
// index      0123456789012345678
String s1 = "programming in java";

println(s1.length());
// 19
println(s1.indexOf("i")); // 8
println(s1.indexOf("gram")); // 3
println(s1.indexOf("hi")); // -1
println(s1.substring(7, 10));
// "min"
println(s1.substring(12));
// "in java"
println(s1.substring(2,3));
// "o"
println(s1.substring(2,2));
// "", empty string
String s2 = s1.substring(10, 17);
// "g in ja"
println(s2.toUpperCase());
// "G IN JA", s2 is still "g in ja"
```

`String` method examples

- Given the following string:

```
// index          01234567890123456789
String book = "Building Java Programs";
```

  - How would you extract the word `"Java"` ?

```
String word = book.substring(9,13);
```

Modifying strings

- Methods like `substring` and `toLowerCase` build and return a new string, rather than modifying the current string.
  - String is **immutable**; once created, its value cannot be changed.

```
String s = "lil bow wow";
s = “snoop dog”;
//”lil bow wow” is discarded and a new String
// object “snoop dog” is created.

s.toUpperCase();
println(s);
// snoop dog, s is not changed
```

Modifying strings

- To modify a variable's value, you must reassign it:

```
String s = "lil bow wow";
s = s.toUpperCase();
println(s);
// LIL BOW WOW
```

`String` test methods

Method	Description
`equals(`**`str`**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**`str`**`)`	whether two strings contain the same characters, ignoring upper vs. lower case

```
String a = "hello", b = "HELLO";
println(a.equals(b)); // false
println(a.equalsIgnoreCase(b)); // true
```

Comparing strings

- Avoid using == to compare objects, including Strings.
  == compares objects by *references* , so it often gives `false` even when two `String`s have the same letters. **Use equals instead.**

```
String a = "hi"; //String literal
String b = "hi";
String c = new String("hi");
println(a == b);
//true, to save on memory, a and b reference the
// same String
println(a == c); //false, a and c reference
// different Strings.
println(a.equals(c));
//true, a and c have the same values.
```

Substring Methods

The following methods are useful. Learn how to use them properly!

```
length()
indexOf(String a)
substring(int index)
substring(int index1, int index2)
equals(String a)
```

k

Substring Methods

String school = “Hogwarts: School for Pigs”;

String headmaster = “Brad Pig”; //
String teacher = “Kevin Bacon”;

String gf = “Piggy Azalea”;

String fav = “Hamlet”;

String favCharsinStarWars = “Ham Solo and Chewbacon”;

Recursion

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 **iterative** version method to print a line of * characters:

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        print("*");
    }
    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:

```
void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        println();
        return; // ends execution of method early
    } else {
        // recursive case; print one more star
        print("*");
        printStars(n - 1);
    }
}
```

Note: In a void method, the empty ”return” statement can be used to ends the execution of a method early. This is sometimes useful in recursive methods.

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
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:

```
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:

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

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

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

– What is this method really doing?

Lab 1: Codingbat

Do the following on codingbat.

In String-1, do:

makeOutWord

firstHalf

theEnd

frontAgain

In String-2, do:

countHi

repeatEnd

catDog

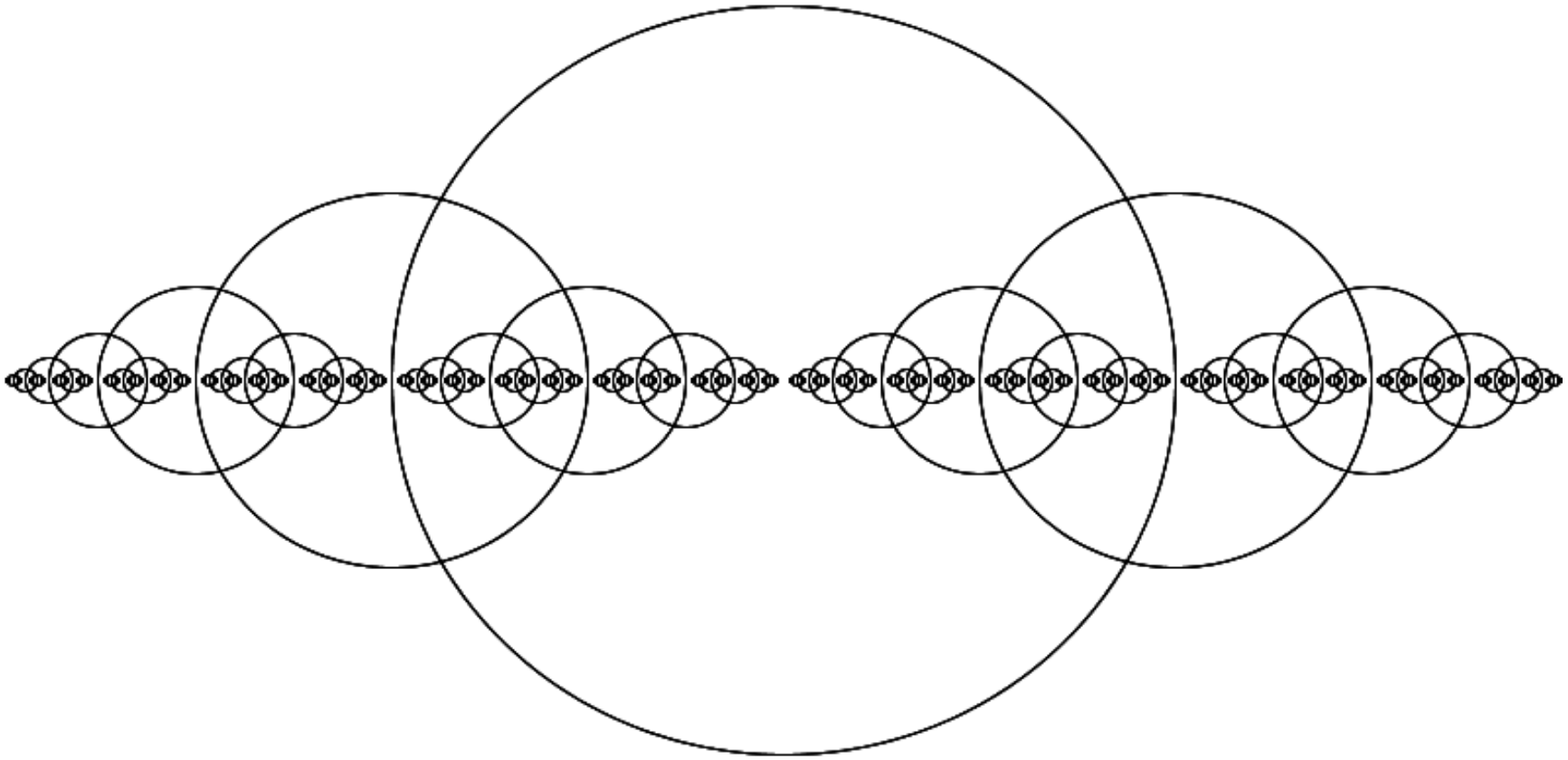
Lab 2: Fractal Circles

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

```
void setup(){
  circle(width/2, width/4, 10);
}
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 2



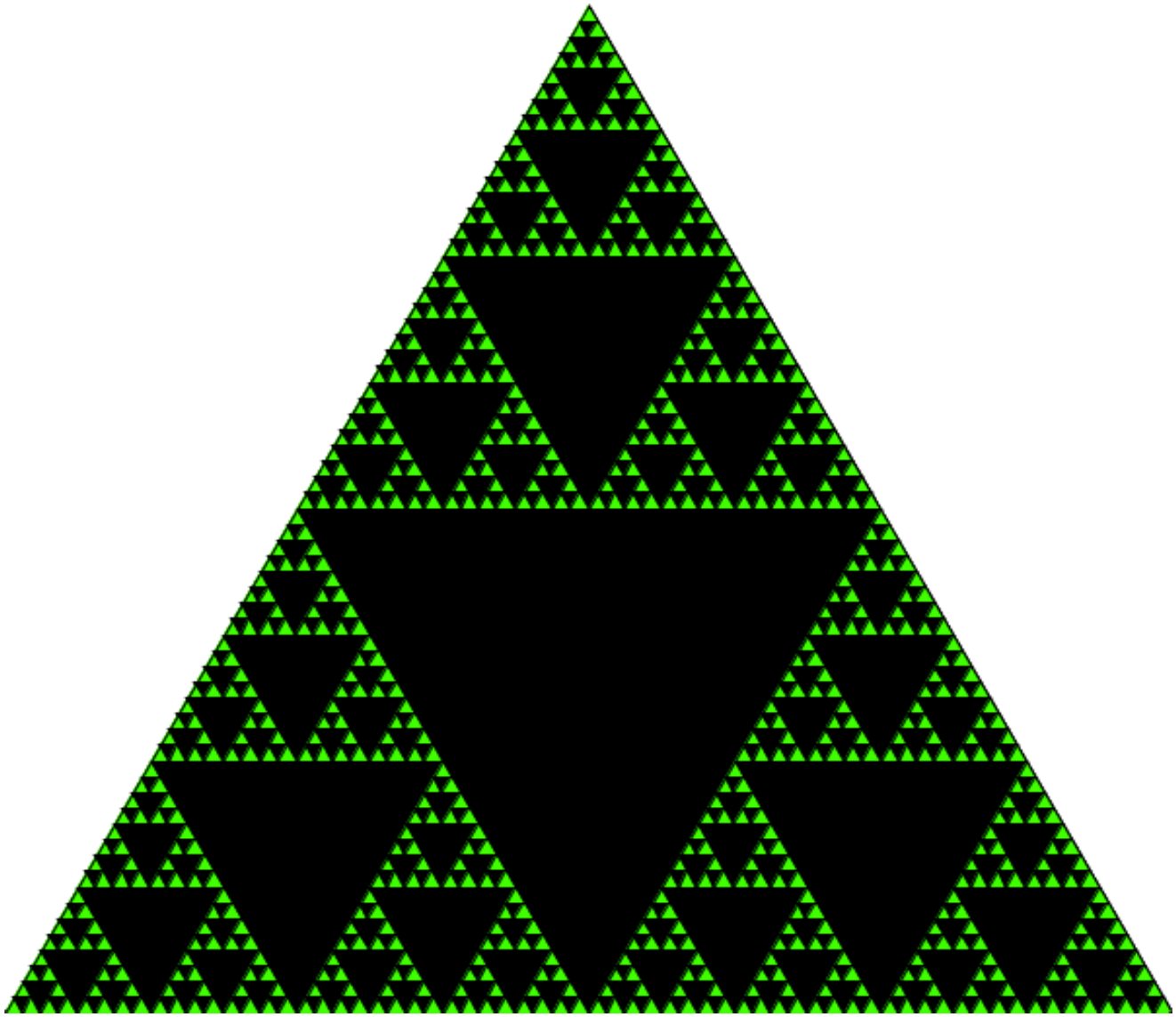
Lab 3: Sierpinski Triangle(Optional)

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

```
void setup(){
  fractralTriangle(width/2,0,0,height,width,height,10);
}

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

Lab 3 Optional



Homework

1) Read and reread these lecture notes.

1) Complete the lab.

References

Part of this lecture is taken from the following book.

1) Stuart Reges and Marty Stepp. Building Java Programs: A Back to Basics Approach. Pearson Education. 2008.