FUNDAMENTALS

BASIC PROGRAMMING MODEL

Ruben Acuña

Fall 2018

2 JAVA SYNTAX REVIEW

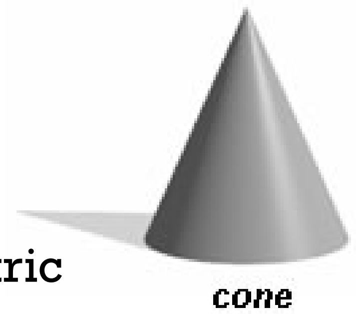
LINEAR SEARCH

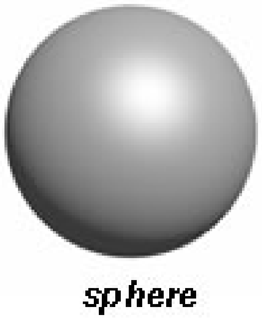
- Problem: Given an array of values, determine if a value is contained in that array.
- Mechanism: Check each element against the value.

- Basic Java Program:
  - Contained in a class.
  - Has a *main()* method.
  - Uses the Java API to read target value.
  - Uses a user defined method to implement search.

```java
/**
 * This program provides an implementation of the linear search algorithm
 * and demonstrates it.
 *
 * @author Acuna
 * @version 1.0
 */
import java.util.Scanner;

public class LinearSearchExample
{
    public static boolean find(int target, int[] pool)
    {
      for(int i = 0; i < pool.length; i++)
        if(pool[i] == target)
          return true;

      return false;
    }

    public static void main(String args[])
    {
        Scanner scanner = new Scanner(System.in);
        int[] data = {4, 45, 8, 1, 3, 3, 22, 9};
        int target;

        System.out.println("What is the target number?");
        target = scanner.nextInt();
        if(find(target, data))
          System.out.println("Found.");
        else
          System.out.println("Missing.");
    }
```

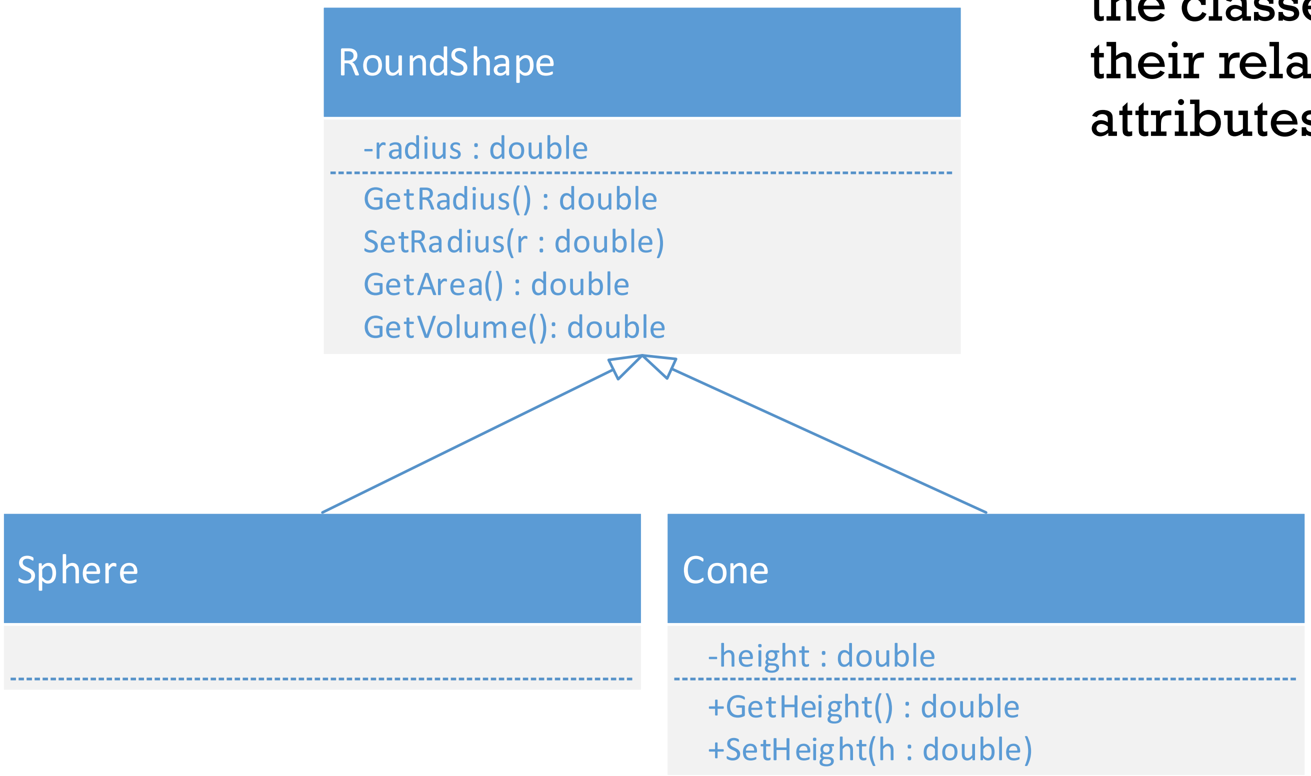
SHAPE LIBRARY





- Problem: Need a way to define geometric shapes in a program.
- In particular, shapes with a radius (*state*).
- Shapes should also be able to compute area and volume (exhibit *behavior*).
- One way to accomplish this aim is to implement shapes as **classes**.
- Organization of the classes should follow their concept(s) in terms of OOP principles.

SHAPE UML



- One standard way to model a system is UML.
- A class diagram describes the classes in a program, their relationships, and attributes/operations.

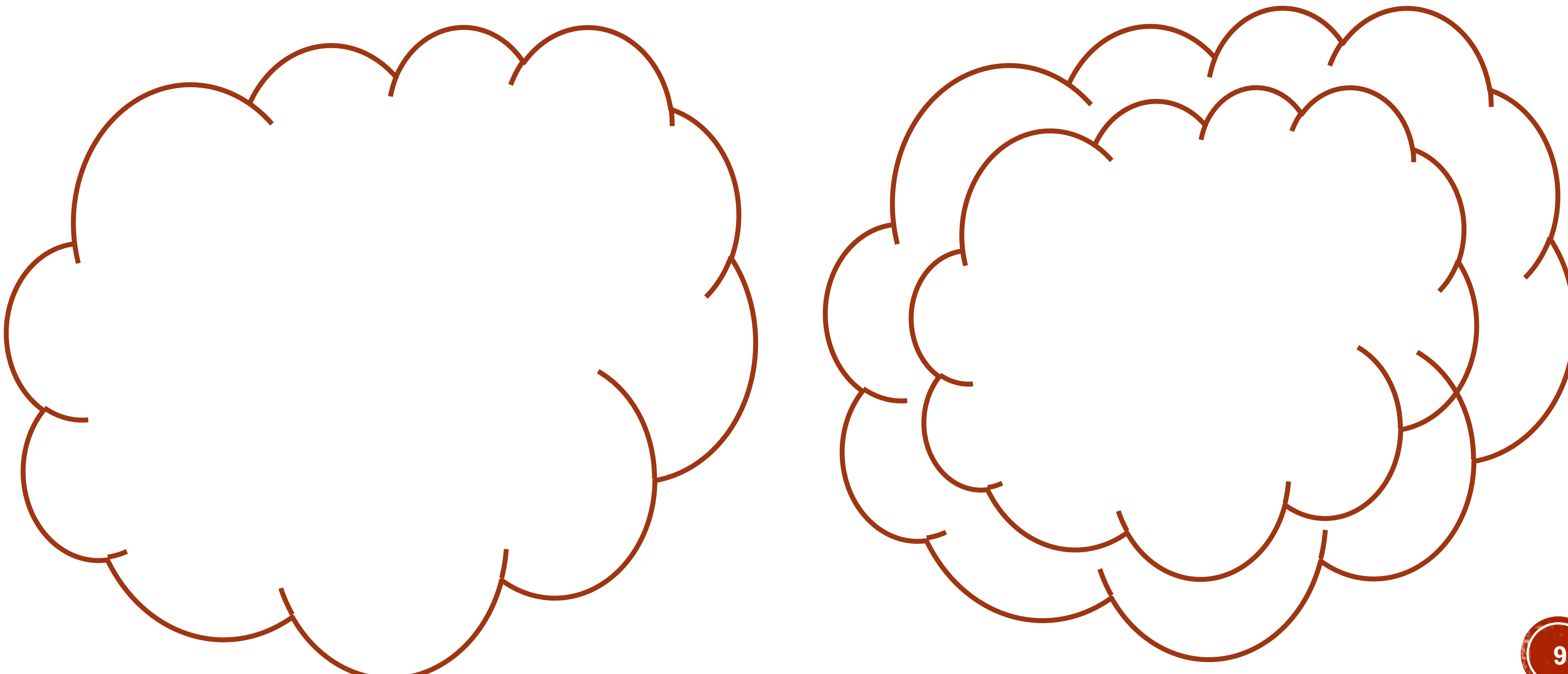
```java
public abstract class RoundShape
{
    private double radius;

    public RoundShape(double r)
    {
        radius = r;
    }

    public double GetRadius()
    {
        return radius;
    }

    public void SetRadius(double r)
    {
        radius = r;
    }

    public abstract double GetArea();
    public abstract double GetVolume();
}
```

```java
public class Cone extends RoundShape
{
    private double height;

    public Cone(double r, double h)
    {
        super(r);
        height = h;
    }

    public double GetHeight()
    {
        return height;
    }

    public void SetHeight(double r)
    {
        height = r;
    }

    public double GetArea()
    {
        return Math.PI * GetRadius() * (GetRadius()
    }

    public double GetVolume()
    {
        return Math.PI * Math.pow(GetRadius(), 2) *
    }

    String ToString()
    {
        return "A cone of radius " + GetRadius() + "
    }
}
```

8 RECURSION

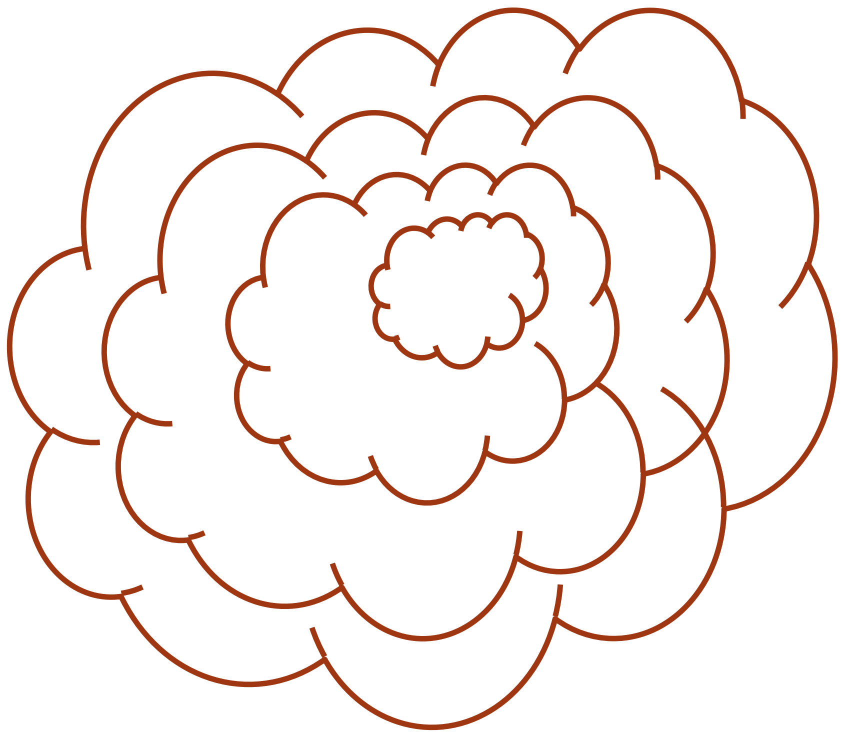
THE RECURSION CONCEPT

Suppose I want to draw a puffy cloud. Well, I can use PowerPoint's handy cloud tool. Only, it's not really puffy. So, I copy the first cloud, shrink it, and move it into the original cloud. That's okay, but that inner cloud doesn't look puffy. Maybe I can repeat the copy…



THE RECURSION CONCEPT

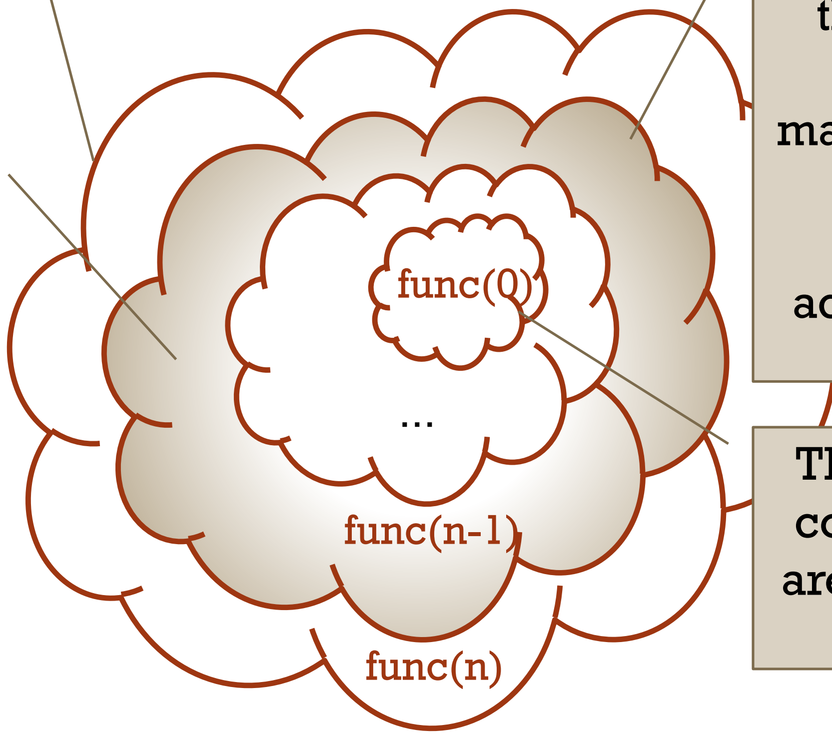
… and so I do. And again. And again. Until I've ended up shrinking the smaller cloud so much, I really don't need another one. So, I built a cloud out of a cloud! This is recursion.



RECURSION OVERVIEW

A full size problem (n) contains a sub-problem (n-1).

Each time the problem is analyzed, some part of the solution is computed and is combined with the solution for the sub-problem, to form the solution for the whole.



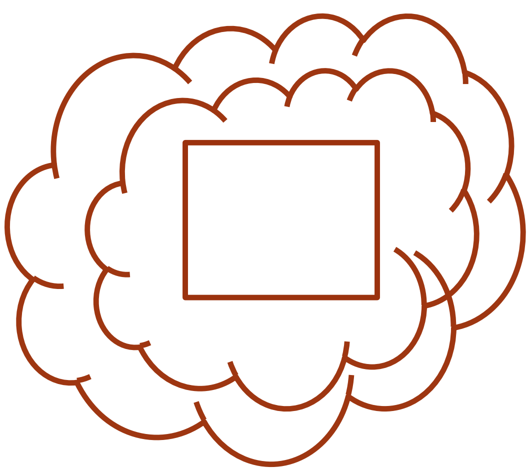
The sub-problem is the same type of problem as the main problem but is easier to solve. It is "smaller" according to a size metric.

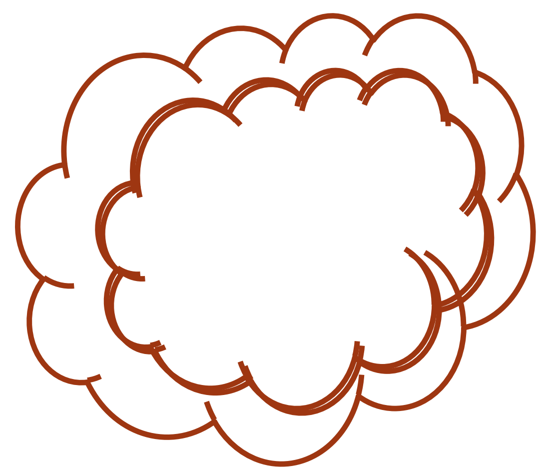
The sub-problems continue until they are small enough to be trivial.

THE KEYS TO RECURSION

For all valid inputs, a recursive method must make the problem smaller during each call, and make ***progress*** towards some base case (***termination***).

The type of problem should not change, nor the sub-problem be the same as the original problem.





RECURSIVE METHODS

- Recursive structure is an important concept in computer science. By definition, a method is recursive if it calls itself within the method.

- There is a special type of recursive structure, which calls itself only once and in the last statement. We refer to this as the tail-recursion. (Tail recursion is very similar to a do-while loop.)

- Why do we need recursion?

THE FANTASTIC FOUR APPROACH

SCAFFOLDING THE DESIGN OF RECURSIVE METHODS

1. Formulate the size-n problem.
2. Find the stopping condition and the corresponding return value.
3. Formulate the size-m problem and find m. In many cases, m = n - 1;
4. Construct the solution of size-n problem from size-m problem.

Let's think about implementing **factorial** in terms of these steps.

(Chen and Tsai. *Introduction to Programming Languages*, Kendall Hunt Publishing.)

1. FORMULATE THE SIZE-N PROBLEM

THE FANTASTIC FOUR ABSTRACT OF WRITING RECURSIVE FUNCTIONS

- Like a loop, recursion is necessary only if you want to solve a problem that needs to repeat the same operations.
- We assume the number of iterations is n. In most cases, n is obvious. For example, if we want to compute factorial n!, the size n is already given.
- Formulating the size-n problem, in some cases, is merely choosing a function name and using n as the parameter of the function. Thus, the size-n problem for factorial problem is

  int factorial(int n)

- The return value of the size-n is what the function is supposed to compute, or the value we are looking for. Obviously, in this step, we do not need to design the solution for size-n problem.

2. FIND THE STOPPING CONDITION AND RETURN VALUE

THE FANTASTIC FOUR ABSTRACT OF WRITING RECURSIVE FUNCTIONS

- Like a while-loop, every recursive function starts with checking the stopping condition.
- If the stopping condition is true, we return the corresponding value and exit the function.
- Otherwise, we enter the body of the recursive function.
- In some cases, identifying the stopping condition and corresponding value is trivial. For example, the stopping condition of factorial(n) is $n = 0$ and the corresponding value is 1

3. FORMULATE THE SIZE-M PROBLEM

THE FANTASTIC FOUR APPROACH OF WRITING RECURSIVE FUNCTIONS

- The size-m problem is simply the size-n problem with n replaced by m, where $m < n$ is determined by how much we can reduce the size in one step.
- If we can only reduce the problem size by one, m is n-1. For example: factorial(n-1).
- Do not try to define a solution, or the return value in this step! All we need to do here is assume the size-m problem will return a value and use the value, e.g., n*factorial(n-1), to get us to the stopping condition.

- The problem may be resized in many different ways:
  - For the mergesort, m = $\frac{1}{2}$ n.
  - For the maze homework, m = $\frac{1}{4}$ n.

4. CONSTRUCT THE SOLUTION OF SIZE-N PROBLEM

THE FANTASTIC FOUR APPROACH OF WRITING RECURSIVE FUNCTIONS

- In this step, we will use the assumed solution or the return value for size-m or size-(n-1) problem to construct the solution of the size-n problem.
- This step is application-specific. In the case of factorial, the solution of the size-n problem is

  n*factorial(n-1);

- Sometimes, we need to use the return values of multiple size-m problems, where $0 \leq m < n$ to construct the solution of size-n problem.

PUTTING FACTORIAL TOGETHER

```
int factorial(int n) {
        if(n == 0)
                return 1;
        else
                return n*factorial(n-1);
}



// Mathematical Definition:
// 1! = 1
// N! = N * (N-1)!
```

MESSING WITH FACTORIAL

COMPUTING THE NTH FIBONNACI NUMBER

1. Formulate the size-n problem.

   long fib(int n)

2. Find the stopping condition and the corresponding return value.

   if(n == 0) return 0;

   if(n == 1) return 1;

3. Formulate the size-m problem and find m. In many cases, m = n - 1;

   fib(n-1)

   fib(n-2)

4. Construct the solution of size-n problem from size-m problem.

   fib(n-1) + fib(n-2)

THE NTH FIBONNACI NUMBER

```
public static long fib(int n) {

    if(n == 0)
        return 0;

    if(n == 1)
        return 1;

    return fib(n-1)+ fib(n-2);
}
```

At this point, recursion probably looks easy. And not that useful. Ha!

As a general rule, recursion is very useful whenever you are faced with a problem you do not know how to solve.

Due to the nature of our world, many problems have a naturally recursive structure, which a recursion algorithm is able to leverage.

PRINTING A LIST

1. Formulate the size-n problem.
   void displayList(LinearNode node)

2. Find the stopping condition and the corresponding return value.
   if(node == null) and return;

3. Formulate the size-m problem and find m. In many cases, m = n - 1;
   displayList(node.getNext());

4. Construct the solution of size-n problem from size-m problem.
   System.out.println(node); displayList(node.getNext());

PRINTING A LIST

```
//iterative
public static void
displayList(LinearNode node) {
  LinearNode iter = node;
  while(iter != null) {
    System.out.println(iter);
    iter = iter.getNext();
  }
}
```

```
//recursive
public static void displayList(LinearNode
node) {
  if(node != null) {
    System.out.println(node);
    displayList(node.getNext());
  }
}
```

RECURSIVE DATA STRUCTURES

Here's a thought: we've made programs recursive, so might we make data recursive too?

```
public class Thing {
    public Thing thing2;
    private int alongfortheride;

    public Thing(Thing t2, int i) {
        thing2 = t2;
        alongfortheride = i;
    }
}
```

This is the fundamental idea behind the "linked list" data structure.

- This class is self-referential – another form of recursion.
- This makes a list a recursive data structure.
- Made possible by references.