The Head TAs Like Cookies

• They would each like to have one of these cookies:

• How many ways can they distribute the cookies among themselves?
  o the first HTA who picks has four choices
  o only three choices are left for the second HTA
  o two choices left for the third HTA
  o the last HTA has to take what remains (poor Nick!)
The Head TAs Like Cookies

• Thus we have 24 different ways the HTAs can choose cookies (4! = 4 x 3 x 2 x 1 = 24)

• What if we wanted to solve this problem for all CS15 TAs? All CS15 students?
Factorial Function

• Model this problem mathematically:
  factorial (n!) calculates the total number of unique permutations of n items

• Small examples:
  1! = 1
  2! = 2*1 = 2
  3! = 3*2*1 = 6
  4! = 4*3*2*1 = 24
  5! = 5*4*3*2*1 = 120

• **Iterative** definition:  n! = n * (n-1) * (n-2) * … * 1

• **Recursive** definition:  n! = n * (n-1)! for n>0
  and 0! = 1
Recursion (1/2)

• Models problems that are self-similar
  o decompose a whole task into smaller, simpler sub-tasks that are similar
  o thus, each subtask can be solved by applying a similar technique

• Whole task solved by combining solutions to sub-tasks
  o special form of divide and conquer
Recursion (2/2)

• Task is defined in terms of itself
  o in Java, modeled by method that calls itself, but each time with simpler case of the problem
  o Java will bookkeep each invocation of the same method just as it does for nested methods that differ, so there is no confusion
  o requires base case (case simple enough to be solved directly, without recursion) to end recursion; otherwise infinite recursion and StackOverflow Exception
  o what is the base case of the factorial problem?
  o Often you combine the results from the separate invocations
Factorial Function Recursively (1/2)

- Recursive algorithm
  - the factorial function is not defined for negative numbers
    - the first conditional checks for this precondition
    - it is good practice to document and test preconditions (see code example)
  - Number of times method is called is called depth of recursion
    - what is depth of (4!)?

```java
public class RecursiveMath{
    //instance variables, other code elided
    public long factorial (int num) {
        if (num < 0){
            System.out.println("Input must be > 0");
            return -1; // return -1 for invalid input
        }
        long result = 0;
        if (num == 0){ // base case: 0! = 1
            result = 1;
        } else{ //general case
            result = num * this.factorial(num - 1);
        }
        return result;
    }
}
```
Factorial Function Recursively (2/2)

4! = factorial(4) = 1
(2*1) 3*(2*1) 4 *(3* (2*1)) = 24
If you want to know more about recursion...
Towers of Hanoi

- Game invented by French mathematician Edouard Lucas in 1883
- **Goal**: move tower of $n$ disks, each of a different size, from left-most peg to right-most peg
- **Rule 1**: no disk can be placed on top of a smaller disk
- **Rule 2**: only one disk can be moved at once
Pseudocode for Towers of Hanoi (1/2)

• Try solving for 5 non-recursively…
• One disk:
  o move disk to final pole
• Two disks:
  o use one disk solution to move top disk to intermediate pole
  o use one disk solution to move bottom disk to final pole
  o use one disk solution to move top disk to final pole
• Three disks:
  o use two disk solution to move top disks to intermediate pole
  o use one disk solution to move bottom disk to final pole
  o use two disk solution to move top disks to final pole
Pseudocode for Towers of Hanoi (2/2)

• In general (for \( n \) disks)
  - use \( n-1 \) disk solution to move top disks to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use \( n-1 \) disk solution to move top disks to final pole

• Note: can have multiple recursive calls in a method
Lower level pseudocode

// n is number of disks, src is starting pole,  
// dst is finishing pole  
public void hanoi(int n, Pole src, Pole dst){  
  if (n==1) {  
    this.move(src, dst);  
  }  
  else {  
    Pole other = this.otherPole(src, dst);  
    this.hanoi(n-1, src, other);  
    this.move(src, dst);  
    this.hanoi(n-1, other, dst);  
  }  
}

public Pole otherPole(Pole p1, Pole p2){  
  // returns the pole that is neither p1 nor p2  
}

public void move(Pole src, Pole dst){  
  // take the top disk on the pole src and make  
  // it the top disk on the pole dst  

• That’s it! otherPole and move are fairly simple methods, so this is not much code.  
• But try hand simulating this when n is greater than 4. Whoo boy, is it tough!  
• Iterative solution far more complex, and much harder to understand
Call Out the Turtles (1/2)

- Fractals: branch of mathematics developed by mathematician Benoit Mandelbrot whose principle characteristic is self-similarity - natural for recursion

- Many examples of simpler, non-fractal, but still self-similar shapes composed of smaller, simpler copies of some pattern
  - spiral, tree, and snowflake

- We can draw these using Turtle graphics
Call Out the Turtles (2/2)

- Let’s start with the simplest: a spiral

  The spiral starts at a particular point. It is made of successively shorter lines, each line at a given angle to the previous one. The user can specify the length of the first side (the longest one), the spiral’s angle, and the amount by which to decrement the spiral’s side in each step.
Designing **Spiral** Class (1/2)

- Spiral class defines single draw method
  - uses turtle to draw, so class needs reference to turtle instance

- From spec, parameters to control its properties:
  - position at which spiral starts is turtle’s position
  - length of spiral’s starting slide
  - angle between successive line segments
  - amount to change length of spiral’s side at each step
  - **Note:** this info is passed to each invocation of recursive method, so next method call depends on previous one
public class Spiral {
    private Turtle _turtle;
    private double _angle;
    private int _lengthDecrement;
    public Spiral(Turtle myTurtle, double myAngle, int myLengthDecrement) {
        _turtle = myTurtle;
        _angle = myAngle;
        _lengthDecrement = 1; // default handles bad parameters
        if (myLengthDecrement > 0){
            _lengthDecrement = myLengthDecrement;
        }
        // draw method defined soon...
    }
}

Designing Spiral Class (2/2)
Drawing Spiral

• First Step: Move turtle forward to draw line and turn some degrees. What’s next?
• Draw smaller line and turn! Then another, and another…
Sending Recursive Messages (1/2)

- **draw** method uses turtle to trace spiral
- How does **draw** method divide up work?
  - draw first side of spiral
  - then draw smaller spiral

```java
public void draw(int sideLen) {
    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```
Sending Recursive Messages (2/2)

- What is the base case?
  - when spiral too small to see, conditional stops method so no more recursive calls made
  - since side length must approach zero to “bottom out” the recursion, i.e., reach the base case, recursive calls to draw pass smaller side length each time

```java
public void draw(int sideLen) {
    // base case: spiral too small to see
    if (sideLen <= 3) {
        return;
    }

    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```
Recursive Methods

• We are used to seeing a method call other methods, but now we see a method calling itself

• Method must handle successively smaller versions of original task
Method’s Variable(s)

- As with separate methods, each invocation of the method has its own copy of parameters and local variables, and shares access to instance variables.
- Parameters let method invocations (i.e., successive recursive calls) “communicate” with, or pass info between, each other.
- Java’s record of current place in code and current values of parameters and local variables is called the activation record.
  - With recursion, multiple activations of a method may exist at once.
  - At base case, as many exist as depth of recursion.
  - Each activation of a method is stored on the activation stack (you’ll learn about stacks soon).
Spiral Activation

```java
draw(int sideLen)
```

Initial value of sideLen: 25
Length decrement: 11

```
draw Activation
int sideLen = 25
```

```
draw Activation
int sideLen = 14
```

```
draw Activation
int sideLen = 3
```

activation of draw method

recursion unwinds after reaching base case

activation of draw method

activation of draw method

Initial value of sideLen: 25
Length decrement: 11
Loops vs. Recursion

• Spiral uses simple form of recursion
  o each sub-task only calls on one other sub-task
  o this form can be used for the same computational tasks as iteration
  o loops and simple recursion are computationally equivalent in the sense of producing the same result, if suitably coded (not necessarily the same performance, though - looping is generally more efficient)
Iterative Spiral

- Iteration often more efficient in Java because recursion takes more method calls (each activation record takes up some of the computer’s memory)
- Recursion more concise and more elegant for tasks that are “naturally” self-similar (Towers of Hanoi is a horror to code non-recursively!)

```java
public void drawIteratively(int sideLen){
    while(sideLen > 3){
        _turtle.forward(sideLen);
        _turtle.left(_angle);
        sideLen -= _lengthDecrement;
    }
}
```
Recursive Binary Tree (1/2)

- The tree is composed of a trunk that splits into two smaller branches that sprout in opposite directions at the same angle.
- Each branch then splits as the trunk did until sub-branch is deemed too small to be seen. Then it is drawn as a leaf.
- The user can specify the length of a tree’s main trunk, the angle at which branches sprout, and the amount to decrement each branch.
Recursive Binary Tree (2/2)

• Compare each left branch to its corresponding right branch
  o right branch is simply rotated copy
• Branches are themselves smaller trees!
  o branches are themselves smaller trees!
    • branches are themselves smaller trees!
    • ...
• Our tree is self-similar and can be programmed recursively!
  o base case is leaf
Self-Similar Images of Trees

Robot with recursively defined body!

Designing the **Tree** Class

- **Tree** has properties that user can set:
  - start position (**myTurtle**’s built in position)
  - angle between branches (**myBranchAngle**)
  - amount to change branch length (**myTrunkDecrement**)

- **Tree** class will define a single **draw** method
  - like **Spiral**, also uses a **Turtle** to draw

```java
public class Tree{
    private Turtle _turtle;
    private double _branchAngle;
    private int _trunkDecrement;
    public Tree(Turtle myTurtle, double myBranchAngle, int myTrunkDecrement){
        _turtle = myTurtle;
        _branchAngle = myBranchAngle;
        _trunkDecrement = 1;
        if(myTrunkDecrement > 0){
            _trunkDecrement = myTrunkDecrement;
        }
        // draw method coming up...
    }
}
```
Tree’s draw Method

• **Base case:** if branch size too small, add a leaf
• **General case:**
  o move _turtle forward
  o orient _turtle left
  o recursively draw left branch
  o orient _turtle right
  o recursively draw right branch
  o reset _turtle to starting orientation
  o back up to prepare for next branch

```java
private void draw(int trunkLen) {
    if (trunkLen <= 0) {
        this.addLeaf();
    } else {
        _turtle.forward(trunkLen);
        _turtle.left(_branchAngle);
        this.draw(trunkLen-_trunkDecrement);
        _turtle.right(2*_branchAngle);
        this.draw(trunkLen-_trunkDecrement);
        _turtle.left(_branchAngle);
        _turtle.back(trunkLen);
    }
}
```

creates a leaf at the current location
Recursive Snowflake

- Invented by Swedish mathematician, Helge von Koch, in 1904; also known as Koch Island
- Snowflake is created by taking an equilateral triangle and partitioning each side into three equal parts. Each side’s middle part is then replaced by another equilateral triangle (with no base) whose sides are one third as long as the original.
  - this process is repeated for each remaining line segment
  - the user can specify the length of the initial equilateral triangle’s side
Snowflake’s **draw** Method

- Can draw equilateral triangle iteratively
- **draw** draws the snowflake by drawing smaller, rotated triangles on each side of the triangle
- **for** loop iterates 3 times
- Each time, calls the **drawSide** helper method (defined in the next slide) and reorients `_turtle` to be ready for the next side

```java
public void drawTriangle(int sideLen) {
    for (int i = 0; i < 3; i++) {
        _turtle.forward(sideLen);
        _turtle.right(120.0);
    }
}

public void drawSnowFlake(int sideLen){
    for(int i = 0; i < 3; i++){
        this.drawSide(sideLen);
        _turtle.right(120.0);
    }
}
```
Snowflake’s **drawSide** method

- **drawSide** draws single side of a recursive snowflake by drawing four recursive sides
- **Base case**: simply draw a straight side
- **MIN_SIDE** is a constant we set indicating the smallest desired side length
- **General case**: draw complete recursive side, then reorient for next recursive side

```java
private void drawSide(int sideLen){
    if (sideLen < MIN_SIDE){
        _turtle.forward(sideLen);
    } else{
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.right(120.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
    }
}
```
Hand Simulation

MIN_SIDE: 20
sideLen: 90

1) Call `draw(90)`, which calls `drawSide(90)`, which calls `drawSide(30)`, which calls `drawSide(10)`. Base case reached because 10 < MIN_SIDE

2) `drawSide(10)` returns to `drawSide(30)`, which tells _turtle to turn left 60 degrees and then calls `drawSide(10)` again.

3) `drawSide(10)` returns to `drawSide(30)`, which tells _turtle to turn right 120 and then calls `drawSide(10)` for a third time.

4) `drawSide(10)` returns to `drawSide(30)`, which tells _turtle to turn left 60 degrees and then calls `drawSide(10)` for a fourth time.

After this call, `drawSide(30)` returns to `drawSide(90)`, which reorients _turtle and calls `drawSide(30)` again.
Again: Koch Snowflake Progression

colored triangles added for emphasis only
Indirect Recursion

• Two or more methods act recursively instead of just one
• For example, methodA calls methodB which calls methodA again
• Methods may be implemented in same or different objects
• Can be implemented with more than two methods, too
• Recursion has many more aspects, is central to CS17 and CS19, and has a branch of mathematics/mathematical logic devoted to it: recursive function theory
Announcements

• Please check the spreadsheet for the location of your DoodleJump Design Check

• DoodleJump is a complex project – we strongly encourage you to start early

• Remember: We have permanently moved to Macmillan 117