### Drawing/Animation

- **Coordinate modification**
  - No variables in the shape coordinates
  - Variables added to x and y
  - Trigs involving angle variable added to x and y
  - Scale factor multiplied to x and y

- **Transformations**
  - No variables in the shape coordinates
  - Shape is drawn centered on (0, 0)
  - Translate
  - Rotate
  - Scale

### Program Structure

Class Leaf{
  • fields: x, y, size, angle, spin etc
  • display()
    pushMatrix();
    translate(x, y);
    rotate(angle);
    scale(size);
    // drawing...
    popMatrix();
  • move(): updates x and y
  • spin(): updates angle}

- Leaf[] leaves = new Leaf[20];
- int idx = 0;
- keyPressed()
  if (key == 's') {
    spin = true;
    //spin = !spin;
  }
- mousePressed()
  leaves[idx] = new Leaf(mouseX, mouseY);
  idx++;

### Factorial

- The factorial of a positive integer N is computed as the product of N with all positive integers less than or equal to N.

\[
4! = 4 \times 3 \times 2 \times 1 = 24
\]

\[
30! = 30 \times 29 \times \ldots \times 2 \times 1 = 26525285981219105863630848000000000
\]

### Factorial - Iterative Implementation

1. void setup() { 
2.   int A = 10;
3.   int B = factorial(5);
4.   println( B );
5. }
6. int factorial(int N) {
7.   int F = 1;
8.   for( int i=N; i>=1; i-- ) {
9.     F = F * i;
10.    }
11.   return F;
12. }
13. return F;
14. }

Trace it.

### Factorial – Recursive Implementation

1. void setup() { 
2.   int A = 10;
3.   int B = factorial(5);
4.   println( B );
5. }
6. int factorial(int N) {
7.   if (N == 1) {
8.     return 1;
9.   } else {
10.      int F = N * factorial(N-1);
11.     return F;
12.   }
13. }

Factorial can be defined in terms of itself.
Last In First Out (LIFO) Stack of Plates

```java
void setup() {
  int A = 10;
  int B = factorial(5);
  println(B);
}

int factorial(int N) {
  if (N == 1) {
    return 1;
  } else {
    int F = N * factorial(N-1);
    return F;
  }
}
```
1. int factorial(int N=5) {
2.   if (N == 1) {
3.     return 1;
4.   } else {
5.     int F = N * factorial(N-1);
6.     return F;
7.   }
8. }

1. void setup() {
2.   int A = 10;
3.   int B = factorial(5);
4.   println(B);
5. }

1. int factorial(int N) {
2.   if (N == 1) {
3.     return 1;
4.   } else {
5.     int F = N * factorial(N-1);
6.     return F;
7.   }
8. }
```plaintext
def factorial(N):
    if N == 1:
        return 1
    else:
        F = N * factorial(N-1)
        return F

void setup() {
    A = 10;
    B = factorial(5);
    println(B);
}
```
Call Stack

The Call Stack keeps track of …

1. all functions that are suspended, in order
2. the point in the function where execution should resume after the invoked subordinate function returns
3. a snapshot of all variables and values within the scope of the suspended function so these can be restored upon continuing execution

What happens if there is no stopping condition, or “base case”?  

```java
void setup() {
  println(F(12));
}

int F(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return F(n-1) + F(n-2);
  }
}
```

What does this do?

Examples

- recursiveCircles
- recursiveTree
- recursiveTreeTransform