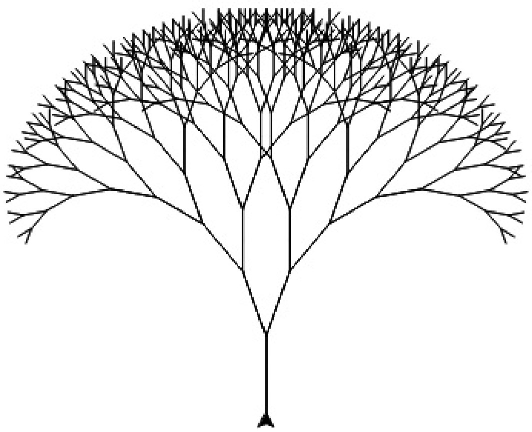
Lab 1

Lab 1 - Recursive Tree



Recursive Tree

Trees can be drawn recursively. Draw a branch. At the end of the branch, draw two smaller branches with one to the left and the other to the right. Repeat until a certain condition is true. This program will walk you through drawing a tree in this way.

Start by declaring a turtle object `t`, and define the method `recursiveTree`. This method should take three parameters, `branchLength`, `angle`, and `t`.

```java
public class RecursiveTree {
  public static void main(String[] args) {

    //add code below this line

    Turtle t = new Turtle(0, 0);

    //add code above this line
  }

  //add method definitions below this line

  /**
  * @param integer branchLength
  * @param integer angle
  * @param Turtle t
  * @return draws a branch of the tree
  */
  public static void recursiveTree(int branchLength, int angle,
        Turtle t) {

  }



  //add method definitions above this line
}
```

The base case for this method is a bit different. In previous examples, if the base case is true a value was returned. The method `recursiveTree` does not return a value, it draws on the screen. So the base case will be to keep recursing as long as `branchLength` is greater than some value. Define the base case as `branchLength` as being greater than 5.

```java
//add method definitions below this line

/**
* @param integer branchLength
* @param integer angle
* @param Turtle t
* @return draws a branch of the tree
*/
public static void recursiveTree(int branchLength, int angle,
      Turtle t) {
  if (branchLength > 5) {

  }
}



//add method definitions above this line
```

Start drawing the tree by going forward and turning right. Call `recursiveTree` again, but reduce `branchLength` by 15. The code should run, but the tree will not look like a tree. It looks more like a curve made of series of line segments decreasing in size.

```java
//add method definitions below this line

/**
* @param integer branchLength
* @param integer angle
* @param Turtle t
* @return draws a branch of the tree
*/
public static void recursiveTree(int branchLength, int angle,
      Turtle t) {
  if (branchLength > 5) {
    t.forward(branchLength);
    t.right(angle);
    recursiveTree(branchLength - 15, angle, t);
  }
}
//add method definitions above this line
```

Do not forget to call the `recursiveTree` method and pass in some initial values.

```java
//add code below this line
Turtle t = new Turtle(0, 0);
recursiveTree(45, 20, t);
//add code above this line
```

The next step is to draw the branch that goes off to the left. Since the turtle turned to the right the number of degrees that the parameter `angle` represents, the turtle needs to turn to the left twice the degrees of `angle`. Turning to the left `angle` will put the turtle back at its original heading. The turtle needs to go further to the left. Then draw another branch whose length is reduced by 15.

```java
//add method definitions below this line

/**
* @param integer branchLength
* @param integer angle
* @param Turtle t
* @return draws a branch of the tree
*/
public static void recursiveTree(int branchLength, int angle,
      Turtle t) {
  if (branchLength > 5) {
    t.forward(branchLength);
    t.right(angle);
    recursiveTree(branchLength - 15, angle, t);
    t.left(angle * 2);
    recursiveTree(branchLength - 15, angle, t);
  }
}
//add method definitions above this line
```

The tree is looking better, but there are two more things that need to be done. First, put the turtle back to its original heading by turning right `angle` degrees. Then go backwards the length of the branch. Call the `recursiveTree` method to draw a tree.

```java
//add method definitions below this line

/**
* @param integer branchLength
* @param integer angle
* @param Turtle t
* @return draws a branch of the tree
*/
public static void recursiveTree(int branchLength, int angle,
      Turtle t) {
  if (branchLength > 5) {
    t.forward(branchLength);
    t.right(angle);
    recursiveTree(branchLength - 15, angle, t);
    t.left(angle * 2);
    recursiveTree(branchLength - 15, angle, t);
    t.right(angle);
    t.backward(branchLength);
  }
}
//add method definitions above this line
```

challenge

What happens if you:

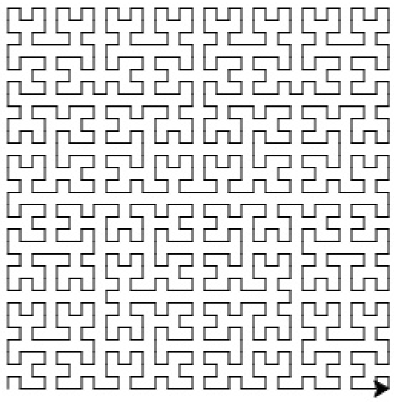
- Increase the branch length when calling `recursiveTree` for the first time?
- Increase and decrease the angle when calling `recursiveTree` for the first time?
- When decreasing `branchLength`, change 15 to something smaller (be sure to change all of the 15's)?
- Change the base case to `if (branchLength > 1)`?
- Rotate the turtle 90 degrees to the left before calling `recursiveTree` for the first time?

▼ **Solution**

```java
public class RecursiveTree {
  public static void main(String[] args) {

    //add code below this line

    Turtle t = new Turtle(0, 0);
    t.left(90);
    t.speed(10);
    recursiveTree(50, 20, t);

    //add code above this line
  }

  //add method definitions below this line

  /**
  * @param integer branchLength
  * @param integer angle
  * @param Turtle t
  * @return draws a branch of the tree
  */
  public static void recursiveTree(int branchLength, int
      angle, Turtle t) {
    if (branchLength > 5) {
      t.forward(branchLength);
      t.right(angle);
      recursiveTree(branchLength - 5, angle, t);
      t.left(angle * 2);
      recursiveTree(branchLength - 5, angle, t);
      t.right(angle);
      t.backward(branchLength);
    }
  }
  //add method definitions above this line
}
```

Lab 2

Lab 2 - The Hilbert Curve



Hilbert Curve

The Hilbert Curve is a fractal, space-filling curve. Start by creating a turtle object, and write the method header for the recursive method `hilbert`. The parameters for the method are the distance the turtle will travel, the rule to be used, an angle (determines how tight the fractal is), depth (how intricate the fractal is), and the turtle object.

```java
public class Hilbert {
  public static void main(String[] args) {

    //add code below this line

    Turtle t = new Turtle(0, 0);

    //add code above this line
  }

  //add method definitions below this line

  /**
  * @param integer dist
  * @param integer rule
  * @param integer angle
  * @param integer depth
  * @param Turtle t
  * @return draws a section of the Hilber Curve
  */
  public static void hilbert(int dist, int rule, int angle, int
        depth, Turtle t) {


  }

  //add method definitions above this line
}
```

The base case for the method is when `depth` is 0. Another way to think about the base case is that if `depth` is greater than 0, keep drawing the fractal. Use `if (depth > 0)` as the base case. Also, there are two rules for the turtle. Ask if `rule` is equal to 1 or if it is equal to 2.

```java
//add method definitions below this line

/**
 * @param integer dist
 * @param integer rule
 * @param integer angle
 * @param integer depth
 * @param Turtle t
 * @return draws a section of the Hilber Curve
 */
public static void hilbert(int dist, int rule, int angle, int
      depth, Turtle t) {
  if (depth > 0) {
    if (rule == 1) {

    }
    if (rule == 2) {

    }
  }
}

//add method definitions above this line
```

If `rule` is equal to 1, then the turtle is going to turn left, recursively call the `hilbert` method with `rule` set to 2, go forward, turn right, recursively call the `hilbert` method with `rule` set to 1, go forward, recursively call the `hilbert` method with `rule` set to 1, turn right, and finally move forward. Because the base case is based on `depth`, it must be reduced by 1 each time the `hilbert` method is called recursively.

```java
    if (rule == 1) {
      t.left(angle);
      hilbert(dist, 2, angle, depth - 1, t);
      t.forward(dist);
      t.right(angle);
      hilbert(dist, 1, angle, depth - 1, t);
      t.forward(dist);
      hilbert(dist, 1, angle, depth - 1, t);
      t.right(angle);
      t.forward(dist);
      hilbert(dist, 2, angle, depth - 1, t);
      t.left(angle);
    }
```

If rule is equal to 2, then the code is almost the inverse of when rule is equal to 1. The turtle will still go forward, but left turns become right turns, right turns become left turns, and recursive calls to hilbert will use 2 instead of 1 for the rule parameter (and vice versa).

```
    if (rule == 2) {
      t.right(angle);
      hilbert(dist, 1, angle, depth - 1, t);
      t.forward(dist);
      t.left(angle);
      hilbert(dist, 2, angle, depth - 1, t);
      t.forward(dist);
      hilbert(dist, 2, angle, depth - 1, t);
      t.left(angle);
      t.forward(dist);
      hilbert(dist, 1, angle, depth - 1, t);
      t.right(angle);
    }
```

Finally, call the hilbert method and run the program to see the fractal.

```
    //add code below this line

    Turtle t = new Turtle(0, 0);
    hilbert(5, 1, 90, 5, t);

    //add code above this line
```

▼ **Speeding up the turtle**

The Hilbert Curve can be slow to draw. You can change the speed of the turtle with the following command `t.speed(10);` before calling the hilbert method.

challenge

What happens if you:

- Change the dist parameter?
- Start with the rule parameter as 2?
- Increase or decrease the angle parameter?
- Increase or decrease the depth parameter?

▼ **Solution**

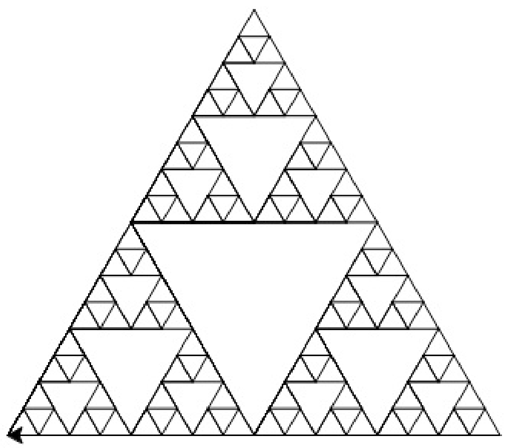
```
  public class Hilbert {
```

```java
public static void main(String[] args) {

  //add code below this line

  Turtle t = new Turtle(0, 0);
  t.speed(10);
  hilbert(5, 1, 90, 5, t);

  //add code above this line
}

//add method definitions below this line

/**
* @param integer dist
* @param integer rule
* @param integer angle
* @param integer depth
* @param Turtle t
* @return draws a section of the Hilber Curve
*/
public static void hilbert(int dist, int rule, int angle,
    int depth, Turtle t) {
  if (depth > 0) {
    if (rule == 1) {
      t.left(angle);
      hilbert(dist, 2, angle, depth - 1, t);
      t.forward(dist);
      t.right(angle);
      hilbert(dist, 1, angle, depth - 1, t);
      t.forward(dist);
      hilbert(dist, 1, angle, depth - 1, t);
      t.right(angle);
      t.forward(dist);
      hilbert(dist, 2, angle, depth - 1, t);
      t.left(angle);
    }
    if (rule == 2) {
      t.right(angle);
      hilbert(dist, 1, angle, depth - 1, t);
      t.forward(dist);
      t.left(angle);
      hilbert(dist, 2, angle, depth - 1, t);
      t.forward(dist);
      hilbert(dist, 2, angle, depth - 1, t);
      t.left(angle);
      t.forward(dist);
      hilbert(dist, 1, angle, depth - 1, t);
```

```
          t.right(angle);
        }
      }
    }

    //add method definitions above this line
  }
```

Lab 3

Lab 3 - Sierpinski Triangle



Sierpinski Triangle

If you start to zoom in on fractals, you will see the same shapes repeat themselves. Fractals are said to be self-similar, which means they can be drawn with recursion. This lab will walk you though drawing a Sierpinski triangle. Start by creating a turtle object. Sierpinski triangles can become quite complex, so increase the turtle's speed to 10 (the maximum).

```
//add code below this line

Turtle t = new Turtle(0, 0);
t.speed(10);

//add code above this line
```

The building block of this fractal is the triangle. Create a method (with parameters for length and a turtle) to draw a triangle. The turtle will be walking all over the screen, so it is important to make sure that the turtle is facing a consistent position before drawing the triangle. `t.setDirection(180)` ensures the turtle is facing to the left.

```java
//add method definitions below this line

/**
 * @param integer length
 * @param Turtle t
 * @return draws a triangle
 */
public static void drawTriangle(int length, Turtle t) {
  t.setDirection(180);
  for (int i = 0; i < 3; i++) {
    t.right(120);
    t.forward(length);
  }
}

//add method definitions above this line
```

Call the `drawTriangle` method to make sure that it words as expected.

```java
  //add code below this line

  Turtle t = new Turtle(0, 0);
  t.speed(10);
  drawTriangle(50, t);

  //add code above this line
```

Look closely at a Sierpinski triangle, and you will see clusters of three triangles that make up clusters of triangles and so forth.

Sierpinski Triangle Evolution

You are now going to create a recursive method that draws this cluster of three triangles. Define the method `sierpinski` that takes `length`, `n`, and `t` as parameters. The base case is if `n` is equal to 1. If so, draw a triangle of size `length`. If `n` is not equal to 1, then you are going to call `sierpinski` again, but with `n-1`. These new triangles need to be in a different position, so move the turtle after drawing each turtle.

```java
//add method definitions below this line

/**
* @param integer length
* @param integer n
* @param Turtle t
* @return draws triangles in the fractal pattern
*/
public static void sierpinski(int length, int n, Turtle t) {
  if (n == 1) {
    drawTriangle(length, t);
  } else {
    sierpinski(length, n - 1, t);
    t.right(120);
    t.forward(length);
    sierpinski(length, n - 1, t);
    t.left(120);
    t.forward(length);
    sierpinski(length, n - 1, t);
    t.forward(length);
  }
}

/**
* @param integer length
* @param Turtle t
* @return draws a triangle
*/
public static void drawTriangle(int length, Turtle t) {
  t.setDirection(180);
  for (int i = 0; i < 3; i++) {
    t.right(120);
    t.forward(length);
  }
}

//add method definitions above this line
```

Finally, replace the `drawTriangle` method call with `sierpinski(50, 1, t)`.

```java
//add code below this line

Turtle t = new Turtle(0, 0);
t.speed(10);
sierpinski(50, 1, t);

//add code above this line
```

challenge

What happens if you:

- Change the method call to `sierpinski(50, 2, t);`?
- Change the method call to `sierpinski(50, 3, t);`?
- Change the method call to `sierpinski(50, 4, t);`?

The triangles are clustered together, but the Sierpinski triangle has larger triangle-shaped voids. An adjustment needs to be made to the distance the turtle moves between calls to the `sierpinski` method. Instead of moving forward the distance of `length`, the turtle will move forward `length * (n-1)`.

```java
/**
* @param integer length
* @param integer n
* @param Turtle t
* @return draws triangles in the fractal pattern
*/
public static void sierpinski(int length, int n, Turtle t) {
  if (n == 1) {
    drawTriangle(length, t);
  } else {
    sierpinski(length, n - 1, t);
    t.right(120);
    t.forward(length * (n - 1));
    sierpinski(length, n - 1, t);
    t.left(120);
    t.forward(length * (n - 1));
    sierpinski(length, n - 1, t);
    t.forward(length * (n - 1));
  }
}
```

Change the `sierpinski` method call to `sierpinski(20, 4, t);`.

```
//add code below this line

Turtle t = new Turtle(0, 0);
t.speed(10);
sierpinski(20, 4, t);

//add code above this line
```

The fractal is getting better, but there are a few areas where the program can be improved. Change the distance the turtle goes forward. Instead of multiplying `length` by `n - 1`, multiply `length` by `2` to the power of `n - 2`. Exponents are represented with `Math.pow`.

```
/**
 * @param integer length
 * @param integer n
 * @param Turtle t
 * @return draws triangles in the fractal pattern
 */
public static void sierpinski(int length, int n, Turtle t) {
  if (n == 1) {
    drawTriangle(length, t);
  } else {
    sierpinski(length, n - 1, t);
    t.right(120);
    t.forward(length * Math.pow(2, n - 2));
    sierpinski(length, n - 1, t);
    t.left(120);
    t.forward(length * Math.pow(2, n - 2));
    sierpinski(length, n - 1, t);
    t.forward(length * Math.pow(2, n - 2));
  }
}
```

challenge

What happens if you:

- Change the `sierpinski` method call to `sierpinski(5, 6, t);`?
- Change the `sierpinski` method call to `sierpinski(5, 8, t);`?

▼ **Solution**

```java
import java.lang.Math;

public class Sierpinski {
  public static void main(String[] args) {

    //add code below this line

    Turtle t = new Turtle(0, 0);
    t.speed(10);
    sierpinski(20, 4, t);

    //add code above this line
  }

  //add method definitions below this line

  /**
  * @param integer length
  * @param integer n
  * @param Turtle t
  * @return draws triangles in the fractal pattern
  */
  public static void sierpinski(int length, int n, Turtle t) {
    if (n == 1) {
      drawTriangle(length, t);
    } else {
      sierpinski(length, n - 1, t);
      t.right(120);
      t.forward(length * Math.pow(2, n - 2));
      sierpinski(length, n - 1, t);
      t.left(120);
      t.forward(length * Math.pow(2, n - 2));
      sierpinski(length, n - 1, t);
      t.forward(length * Math.pow(2, n - 2));
    }
  }

  /**
  * @param integer length
  * @param Turtle t
  * @return draws a triangle
  */
  public static void drawTriangle(int length, Turtle t) {
    t.setDirection(180);
    for (int i = 0; i < 3; i++) {
      t.right(120);
      t.forward(length);
```

```
      }
    }

    //add method definitions above this line
  }
```

Lab Challenge

Lab Challenge

Problem
Write a recursive method called `recursivePower` that takes two integers as parameters. The first parameter is the base and the second parameter is the exponent. Return the base parameter to the power of the exponent.

Expected Output
* If the method call is `recursivePower(5, 3)`, then the function would return `125`
* If the method call is `recursivePower(4, 5)`, then the function would return `1024`

Code Visualizer