



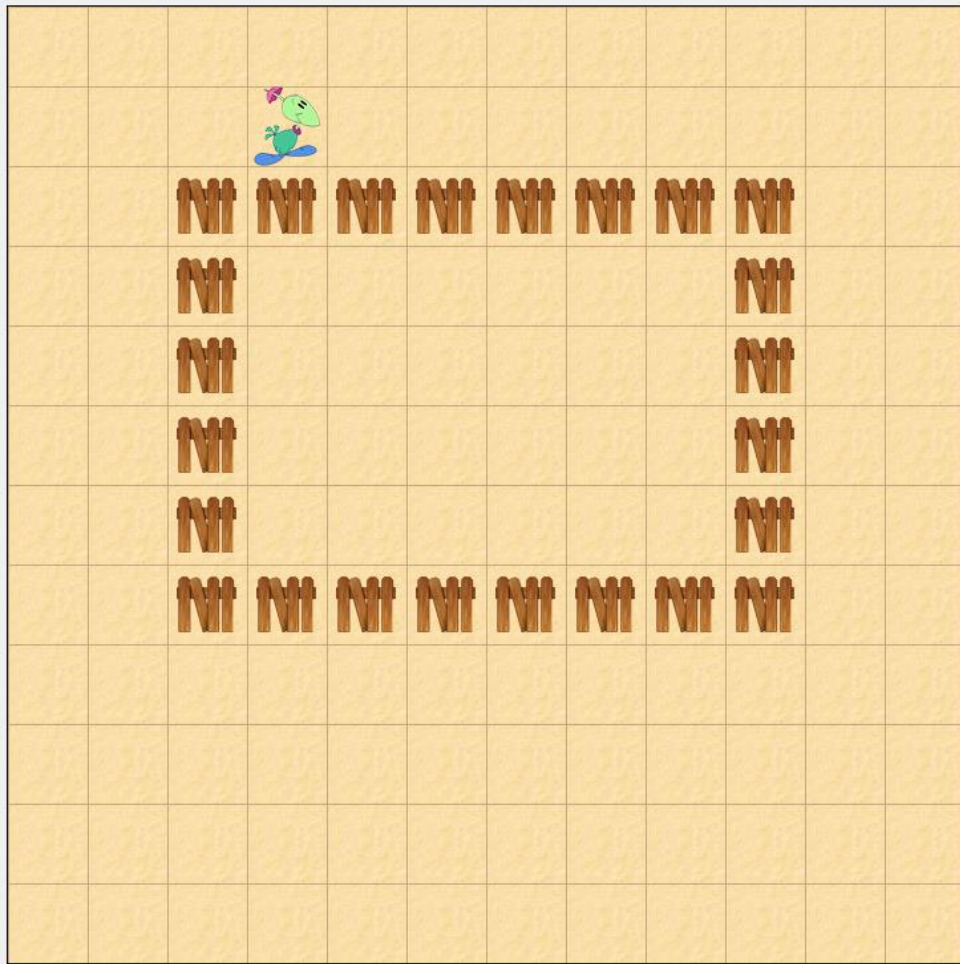
Algorithmic Thinking and Structured Programming (in Greenfoot)

Teachers:

Renske Smetsers-Weeda

Sjaak Smetsers

As 6, part 5.6: Calculating a fenced area

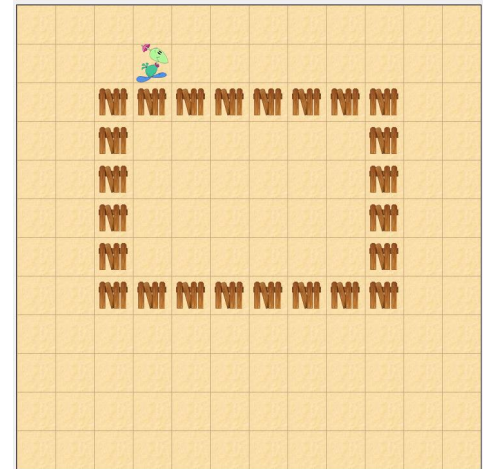


- Execution by pressing the *run* button, no iteration in *act*.

As 6, part 5.6: Calculating a fenced area

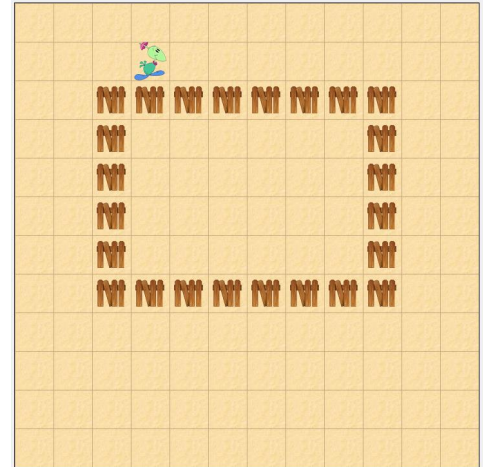
- Step 1: walking around the area
 - Assumption: Mimi is standing next to the area with the fence on the right.
- Solution:

```
public void walkAroundFencedArea() {  
    turnRight();  
    if ( ! canMove() ){  
        turnLeft();  
    }  
    move();  
}
```



As 6, part 5.6: Calculating a fenced area

- Step 2: calculate the area
 - we need 2 variables to keep track of the width and height
 - Are these local variables or instance variables?
- Answer: Instance variables

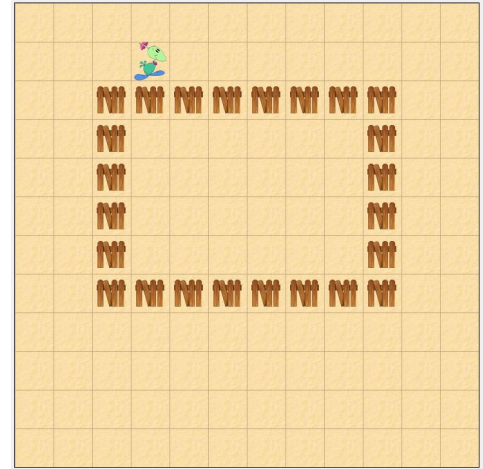


```
public class MyDodo extends Dodo {  
    private int myNrOfStepsTaken;  
  
    private int myAreaWidth;  
    private int myAreaHeight;  
    <...>  
}
```

Instance variables

As 6, part 5.6: Calculating a fenced area

- Instance variables: initialization in the constructor



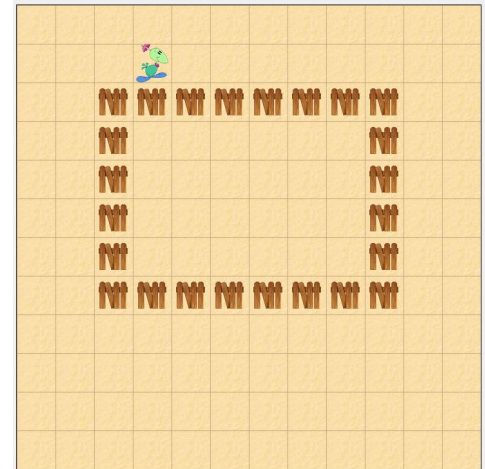
```
public MyDodo( int init_direction ) {  
    super ( init_direction );  
    myNrOfEggsHatched    = 0;  
    myAreaWidth          = 0;  
    myAreaHeight         = 0;  
}
```

Constructor

call to the Dodo constructor

As 6, part 5.6: Calculating a fenced area

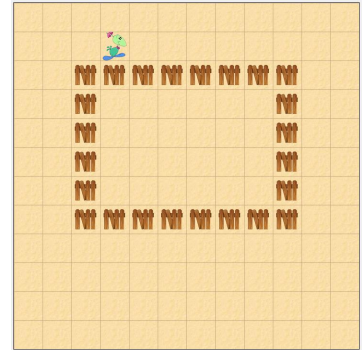
- Adjusting the instance variables.
 - Observation: Mimi does not have to make a complete tour!
 - How does she know that she is done?
 - Answer: as soon as she has computed the width and height.




Calculating a fenced area: Solution

```
public void calculateFencedArea() {
    if ( myAreaWidth == 0 || myAreaHeight == 0 ) {

    } else {
        System.out.println( "The size of the fenced area is "
            + myAreaHeight * myAreaWidth );
        Greenfoot.stop();
    }
}
```



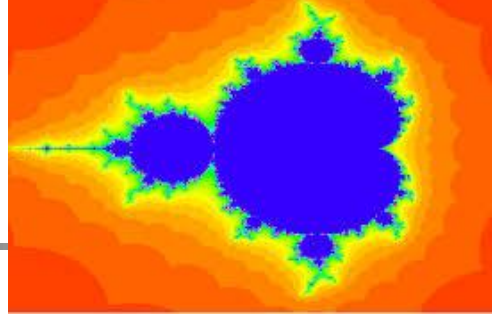


BlueJ

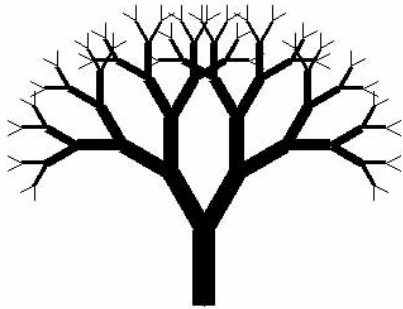


Different programming environment (as opposed to Greenfoot)
Same language: Java

Recursion



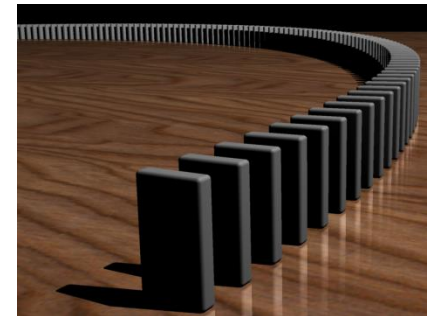
- A smaller part of oneself is embedded in itself
- Many natural phenomena are recursive



(a) Trees



(b) Infinite mirror images



(c) dominos

Sometimes, it is easier to solve a given problem using recursion



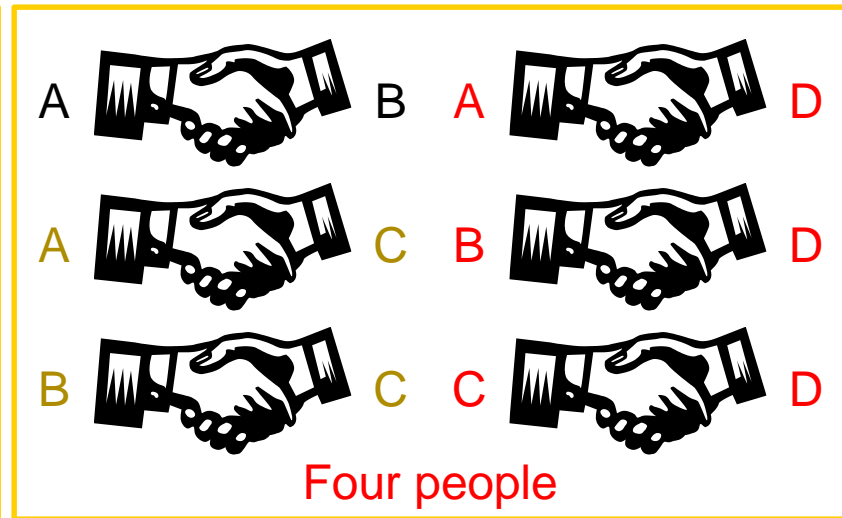
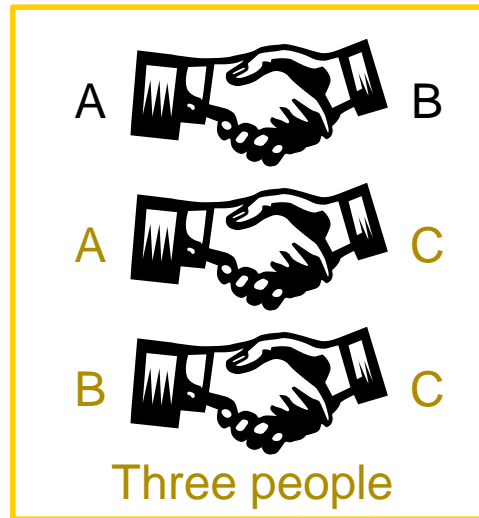
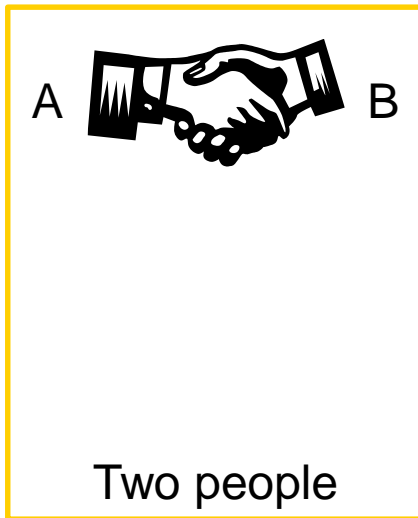
Recursive Definitions

- In a **recursive definition**, an object is defined in terms of itself (but then smaller).
- We can recursively define **sequences, functions, sets, ...**
- **Recursion** is a principle closely related to mathematical induction.

Ex. 1: The handshake problem

Question: There are n people in the room.

If each person shakes hands once with every other person, what will the total number of handshakes be?





Ex. 1: The handshake problem (cont'd)

- There is a trick to know the total number
 - If there are **two** people, only one handshake $h(2) = 1$

let $h(n)$ calculate the number of handshakes needed,
 n 'the number of people' is 2,
 $h(2)$ 'the number of handshakes for 2 people' equals 1.

so $h(2) = 1$



Ex. 1: The handshake problem (cont'd)

- There is a trick to know the total number
 - If there are **two** people, only one handshake $h(2) = 1$
 - If there are **three people**, treat it as having one more person added to the two people, and shakes hands with them (2 extra handshakes) $h(3) = h(2) + 2$

let $h(n)$ calculate the number of handshakes needed,
 n 'the number of people' is 3,

$h(3)$ 'the number of handshakes' for 3 people equals:

- the number of handshakes needed for 2 people, so $h(2)$
- plus two more handshakes, so $+ 2$

so $h(3) = h(2) + 2$



Ex. 1: The handshake problem (cont'd)

- There is a trick to know the total number
 - If there are **two** people, only one handshake $h(2) = 1$
 - If there are **three people**, treat it as having one more person added to the two people, and shakes hands with them (2 extra handshakes) $h(3) = h(2) + 2$
 - If there are **four people**, treat it as having one more person added to the **three people**, and shakes hands with them (3 extra handshakes) $h(4) = h(3) + 3$

let $h(n)$ calculate the number of handshakes needed,
 n 'the number of people' is 4,

$h(4)$ 'the number of handshakes' for 4 people equals:

- the number of handshakes needed for 3 people, so $h(3)$
- plus two more handshakes, so + 3

so $h(4) = h(3) + 3$



Ex. 1: The handshake problem (cont'd)

- There is a trick to know the total number
 - If there are two people, only one handshake $h(2) = 1$
 - If there are **three people**, treat it as having one more person added to the two people, and shakes hands with them (2 extra handshakes) $h(3) = h(2) + 2$
 - If there are **four people**, treat it as having one more person added to the **three people**, and shakes hands with them (3 extra handshakes) $h(4) = h(3) + 3$
- We can **generalize** the total number of handshakes into a formula:

$$\begin{aligned} h(n) &= h(n-1) + (n-1) && \text{if } n \geq 2 \\ h(n) &= 0 && \text{otherwise} \end{aligned}$$



Ex. 2: Factorial function

- Recursion is useful for problems that can be represented by a simpler version of the same problem
- Example: the factorial function

$$6! = 6 * \underbrace{5 * 4 * 3 * 2 * 1}_{5!}$$

We could write:

$$6! = 6 * 5!$$



Ex. 2: Factorial function

In general, we can express the factorial function as follows:

$$n! = n * (n-1)!$$

Is this correct? Well... almost ...

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

$$n! = n * (n-1)! \quad (\text{if } n \text{ is larger than } 1)$$

$$n! = 1 \quad (\text{if } n \text{ is equal to } 1)$$



Recursion

- Recursion is one way to **decompose** a task into smaller subtasks
 - Each of these subtasks is a **simpler example** of the same task
 - The smallest example of the same task has a non-recursive solution

- The factorial function
 - $n! = n * (n-1)!$ (simpler subtask is $(n-1)!$)
 - $1! = 1$ (the simplest example is n equals 1)

How many pairs of rabbits can be produced from a single pair in a year's time?

Assumptions:

- Each new pair of rabbits becomes fertile at the age of one month
- Each pair of fertile rabbits produces a new pair of offspring every month;
- None of the rabbits dies in that year.

How the population develops:

- We start with a single pair of (newborn) rabbits;
- After 1 month, the pair of rabbits become fertile
- After 2 months, there will be 2 pairs of rabbits
- After 3 months, there will be 3 pairs ($2+1=3$)
- After 4 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all ($2+3=5$)).

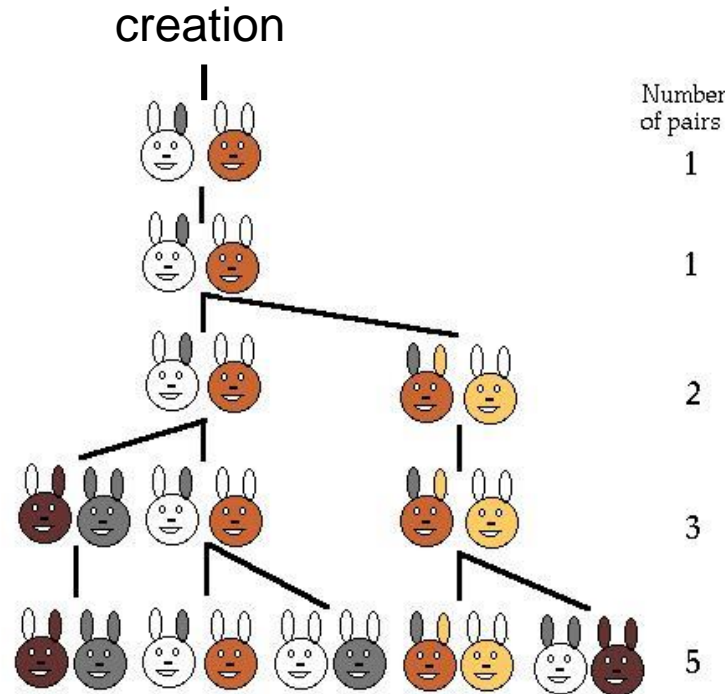


Monthly rabbit population: 1, 1, 2, 3, 5, ...

Population growth in nature

- Leonardo Pisano (nickname: Fibonacci) proposed the sequence in 1202 in *The Book of the Abacus*.

Monthly rabbit population: 1, 1, 2, 3, 5, ...



How many pairs of rabbits can be produced from a single pair in a year's time?

- Can you **generalize** the total number of pairs into a formula?
- Monthly rabbit population: 1, 1, 2, 3, 5, ...



- Reminder. Our handshake formula:



$$\begin{aligned} h(n) &= h(n-1) + (n-1) && \text{if } n \geq 2 \\ h(n) &= 0 && \text{otherwise} \end{aligned}$$



Fibonacci

□ Fibonacci numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each number is the sum of the preceding two

example: $f(2) = f(1) + f(0)$

$$f(3) = f(2) + f(1)$$

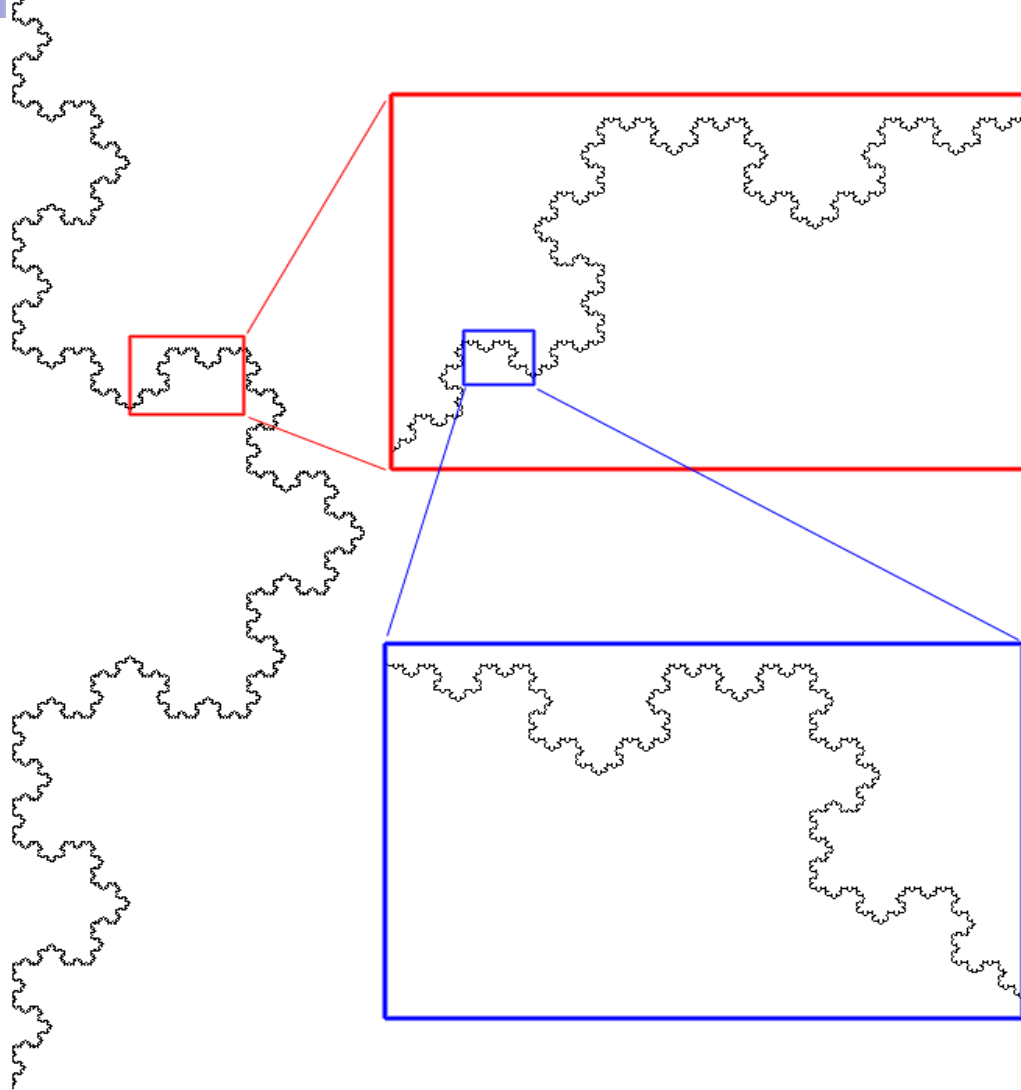
□ Recursive definition:

■ $F(0) = 1$ (Fibonacci number at 0th position)


■ $F(1) = 1$ (Fibonacci number at 1st position)

■ $F(\text{number}) = F(\text{number}-1) + F(\text{number}-2)$

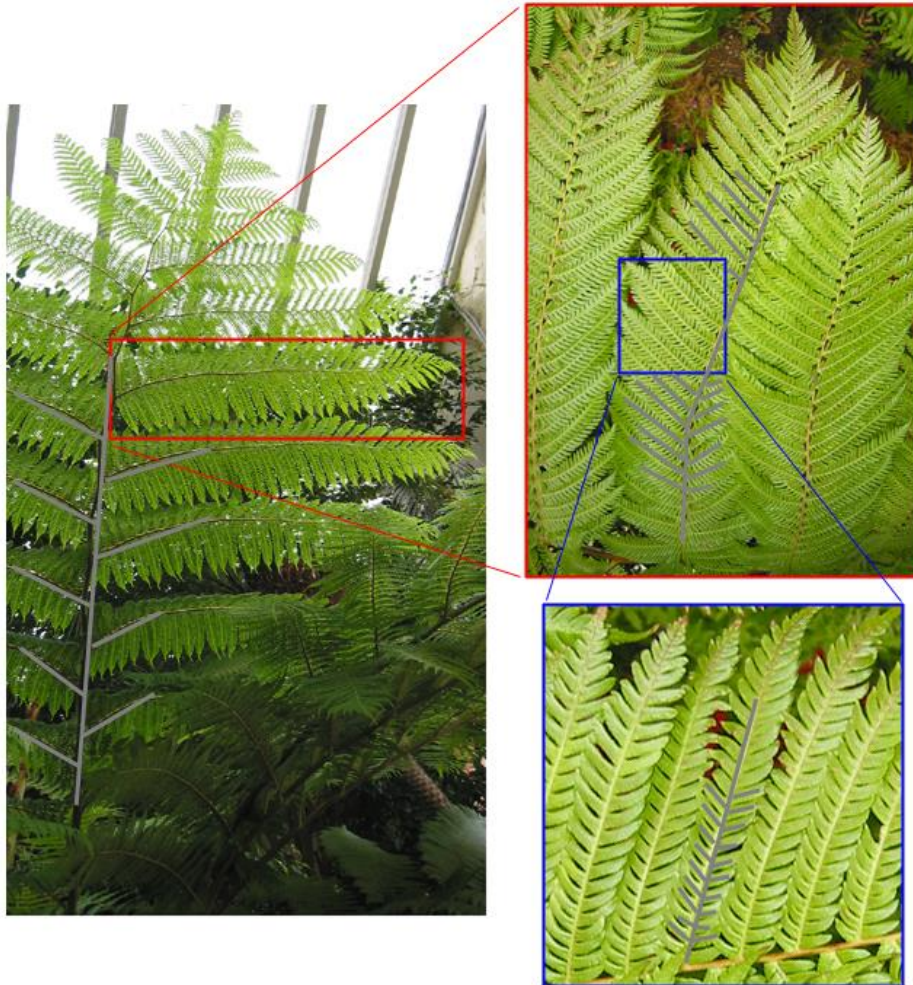
Fractals: self-similar patterns



Self-Similarity in Fractals

- Exact
- Example Koch snowflake curve
- Starts with a single line segment
- On each iteration replace each segment by 
- As one successively zooms in the resulting shape is exactly the same

Self-similarity in Nature





BlueJ and recursion

- ❑ BlueJ is environment (IDE) for Java programming (as an alternative for Greenfoot).
- ❑ In this assignment you will experiment with recursion.

Drawing trees:

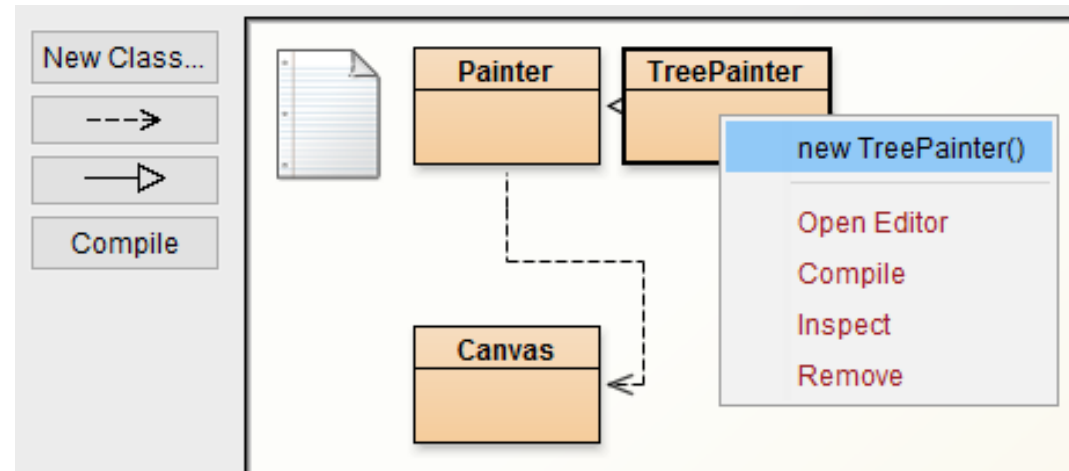
- ❑ Using recursion typically less effort than 'by hand'
- ❑ Recursive definition is the basis for animated movies and games.



Getting started with BlueJ

How to call a tree-drawing method

1. Right-click on the TreePainter class and select 'new TreePainter()'



2. An empty canvas is created. Move it aside (don't click it away).



3. In the bottom of the screen, right-click on the instance you just created:
4. Choose one of the methods to draw a tree.
5. Each time you wish to draw a new tree, repeat the steps above. You can keep multiple canvases open at a time.



Canvas orientation

- ❑ Coordinates are as you are accustomed to in math
(opposed to Greenfoot)
- ❑ Origin (0,0) is in the bottom left corner
- ❑ Always starts facing East
 - After turning 90 degrees (counterclockwise),
pointer faces North



Understanding drawSimpleTree

```
void drawSimpleTree( double length, double beginX , double beginY, double dir )
```

Tinker (“play around with”) assignment:

- Run, view and analyze the code
- Try to figure out how it works.



Calculating coordinates and angles

Method is given `beginX`, `beginY`, `length` and `dir`

Must calculate `endX` and `endY` and `new direction`

Calculate x coordinate for end of branch:

```
double endX = beginX + length * Math.cos ( dir );
```

Calculate y coordinate for end of branch:

```
double endY = beginY + length * Math.sin ( dir );
```

Calculate next angle:

```
dir + bendAngleSimpleTree
```

```
double bendAngleSimpleTree = 22.0/180 * Math.PI;  
(uses 22 degrees and then turns degrees into radians)
```



drawSimpleTree method explained

The first time method is called with the **trunk** information:

```
public void drawSimpleTree() {  
    drawSimpleTree( 180, CANVAS_WIDTH/2, 50, Math.PI/2 );  
}
```

After drawing the **trunk**, the method calls itself **2** times, each time with a **shorter branch** and a **new direction**:

```
void drawSimpleTree( double length, double beginX , double beginY, double dir )  
    ....  
    drawLine( beginX, beginY, endX, endY);  
    double lengthSubTree = length * shrinkFactorSimpleTree; // shrink branch  
    drawSimpleTree ( lengthSubTree, endX , endY, dir + bendAngleSimpleTree );  
    drawSimpleTree ( lengthSubTree, endX , endY, dir - bendAngleSimpleTree );  
}
```

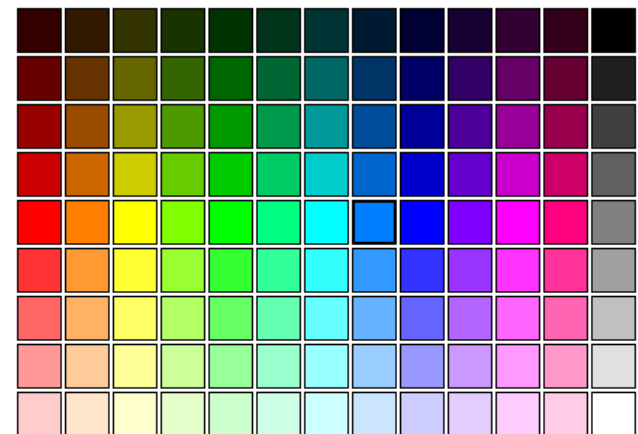
The algorithm **stops** when the branches become too small
(**shorter than length 2**)

drawPurpleTree method explained

More variation:

- Use of **colors**
- Define colors using RGB (Red-Green-Blue) color space

```
setPenColor ( 0, 128, 255 );
```



Hex: # 0080FF

Red: 0

Green: 128

Blue: 255



Tinker assignment:

- Experiment with a different (more **natural**) pen color
- Tip: Google “RGB table”



drawFullBodyTree method explained

More variation for an even more natural look:

- Branch **thickness**

- Algorithm:

- If branch length is long (tree trunk and main branches)
 - Branch is drawn **thick**
- else, the length is short (small branches & leaves)
 - Branch is drawn thin (with minimum of 1 pixel)

Tinker assignment:

- Run, view and analyze the code.
- Experiment with a different length and `treeLengthWidthRatio`



drawMinorRandomTree explained

More variation for an even more natural look:

- Randomness

- `getRandomNumber(60, 90)`

- returns a random int between 60 and 90

- Algorithm:

- Branch length is shrunked by a `shrinkFactor`

- between 60% and 90%

- subtree is drawn

drawNaturalTree

Assignment: Write your own tree method

- Add more variation for a more **natural** look:
 - Combining branch **thickness** and use of **colors**
 - More randomness of angles and lengths
 - Incorporate randomness in colors
 - Use appropriate colors, i.e. different (random) shades of green/brown, but not hot-pink
- Randomness in branches:
 - Occasionally leave out a branch
 - Occasionally draw one branch in front of the other
- .. What else can you draw? (a Christmas tree???)





-
- Write a new method
 - Copy the code from `drawSimpleTree`
 - Add code, inspired from:
 - `drawPurpleTree`
 - `drawFullBodyTree`
 - `drawMinorRandomTree`



Questions?






Wrapping up

- Final test: what to expect (next sheet)
- Final assignment: send us your MyDodo.java file
- Final course survey:
<http://goo.gl/forms/m3TmC32SkE9yHw503>



Test: what to expect

- During testweek
- Theory in assignments 1 through 7
- Similar to the quizzes
- A bit of theory
- Algorithms, flowcharts and code:
 - Designing
 - Analyzing
 - Writing



Thank You!!

And as a final remark:

Thank you all!

We really enjoyed teaching you 😊

After handing in MyDodo.java and passing the final test:

- **You will get a certificate from the RU**
- **Be sure to include this on your CV!!**