

2020 CI401
Introduction to programming

Week 1.09
Algorithms

Dr Roger Evans
Module leader
1st December 2020

Lecture recording and attendance

- This lecture will be recorded and published in the module area
- The focus of recording is on the lecturer, not the audience
- If you are particularly concerned not to be part of the recording, turn off your microphone and camera.
- In addition, lecture attendance is now being routinely recorded (in all modules) to help the School Office monitor engagement
- (This slide is really a reminder to me to start recording and record attendance!)

Module structure (version 3)

Semester 1

Week	Topic	Theme
1.01	Introduction / Hello World	Coding
1.02	Variables, loops and choices	Coding
1.03	Input, more loops and choices	Coding
1.04	Variables and expressions	Coding
1.05	Types, assignment and arrays	Data
1.06	Let's play Top Trumps!	Data
1.07	Objects and methods	OO
1.08	Working with numbers	Data
1.09	Simple Algorithms	Dvp
1.10	Introduction to JavaFX	Dvp
1.11	Simple Animation	Dvp
	Xmas vacation 21 Dec – 8 Jan	
1.12	GUIs using MVC	OO
1.13		

Semester 2

Week	Topic	Theme	Project
2.01	Project topics and assessment	Project	Set
2.02	Simple Inheritance	OO	Lab
2.03	Scope, Visibility and Encapsulation	OO	Lab
2.04	Testing - JUnit	Testing	Lab
2.05	Documentation - Javadoc	Doc	Study
2.06	Collections and generic types	Data	Study
2.07	IO: files and streams	Dvp	Study
	Easter Vacation 29 Mar – 16 Apr		
2.08	Numbers – the computer's view	Data	Study
2.09	Java vs Python		Submit?
2.10	More algorithms – search and sort	Dvp	
2.11	How fast is my code?	Dvp	
2.12	Java 'under the hood'		
2.13	Revision week		Exam ↓

Going online for Christmas!

- From **next week**, all teaching will be online (including labs)
- This is to allow you to go home in time for Christmas (even if you have to self-isolate before you go or after you get home)
- You can still use the labs yourselves, but we won't be teaching in them
- You can connect to our online lab session from the labs if you want to

Module disruption – UCU strike

- The main lecturer's union, the UCU, has called for strike action at Brighton to defend IT staff who are being made redundant at Christmas
- If it goes ahead, this action will cause some disruption to this module (and possibly some of your other modules)
- **Look out for notification of sessions being cancelled because of the strike**
- Because the strike issue could be resolved at any point, these notifications may appear only on the day of the cancellation, so look out for them
- It will be up to University management whether cancelled sessions are made up at a later date
- For further information about the strike and the reasons for it, see <http://blogs.brighton.ac.uk/ucu/it-sackings-strike-student-info/>

Algorithms

Thanks to former colleague Mike Smith for some of the examples used this week

What is an algorithm?

- A step by step process for calculating a solution to a particular problem, or achieving a particular task
- Examples:
 - A recipe for baking a cake
 - Instructions for building flat-pack furniture
 - A knitting pattern
 - Directions – from a passer-by or from Google maps

Algorithms occur in many places

- A recipe for baking a cake
 - Clear list of all ingredients and quantities
 - Clear specification of processes/goals
- Instructions for building flat-pack furniture
 - Pictorial instructions with 'universal' symbols
- A knitting pattern
 - Very precise description of patterns of stitches
- Directions – from a passer-by or from Google Maps
 - Passer-by - vague, dependent on local knowledge, inaccurate
 - Google Maps – very specific, not always based on 'normal' landmarks

Algorithms can have complex structure

- A recipe for baking a cake
 - “Whisk egg whites until stiff”
 - “Bring to the boil, stirring continuously”
 - “Meanwhile, prepare the filling”
- Instructions for building flat-pack furniture
 - Order of steps is not always clearly specified
 - Repeating same action on several parts
- A knitting pattern
 - Often complex sequences of actions and repetitions (in their own ‘programming language’)
- Directions – from a passer-by or from Google Maps
 - Generally a simple sequence of steps
 - May involve “until you reach ...”

Algorithms in computers

- ‘Code’ is designed to be exactly what you need to write algorithms – specific, clear, steps in a process
- It’s boring to say again, but the main components are:
 - Sequence
 - Selection (if, switch stmts)
 - Iteration (for, until stmts)
- We can also use **methods** to break a bigger algorithm into smaller parts
- These are the building blocks for trying to turn a task into an algorithm

A brief aside: Turing machines

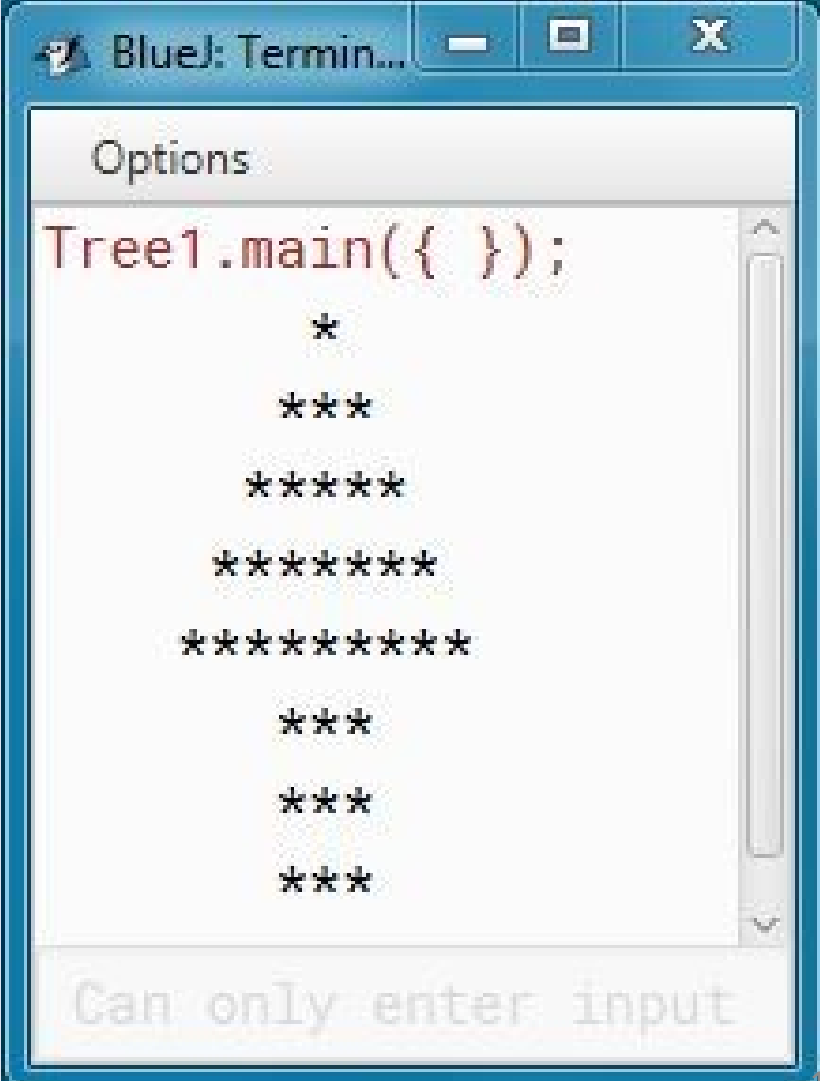
- This set of building blocks doesn't necessarily cover everything you might think of as a way of solving a problem.
- It actually represents the set of problems that can be solved by a **Turing Machine**
- Turing machines are a way of thinking about computing as mathematics proposed by Alan Turing, the famous mathematician and wartime code-breaker.
- Modern computers are examples of **Universal Turing Machines** – these are Turing machines which can simulate any other possible Turing machine. The way we tell a computer which Turing machine to simulate is what we call a **computer program**.
- Turing machines can solve many calculation problems occurring in the real world, but not all possible problems.
- One problem they can't solve is the **Halting problem** – it is not possible to write a program which, given any other program and some input for it, will be able to tell you if that program ever finishes (halts) or not.

Algorithms in computers

- In order to create (computational) algorithms to achieve a particular task, we need to understand the task in terms of these building blocks:
 - think of the task as a **sequence** of steps
 - identify **decision points** where we **select** different paths to take (which may stay separate, or may join up again later in the task)
 - look for **patterns** in the solution to a problem which can be represented by **iteration** (especially important to make our algorithm general-purpose – we don't want an algorithm for sorting 5 things, and then need another one for sorting 6 things, and another for sorting 7 etc...)
- As well as the steps, we need to think about what **information** is used to control the process (make decisions, manage loops etc), which we can store in **variables**.

An example – printing a tree

- **The task:** to print a picture of a tree using just the star symbol on the console output, with a four-space border on the left.
- The task has one parameter, which is the height of the foliage (leaves). So we should be able to print shorter and taller trees



The screenshot shows a BlueJ IDE window titled "BlueJ: Termin...". Inside the window, the "Options" tab is active, displaying the following Java code:

```
Tree1.main({ });
```

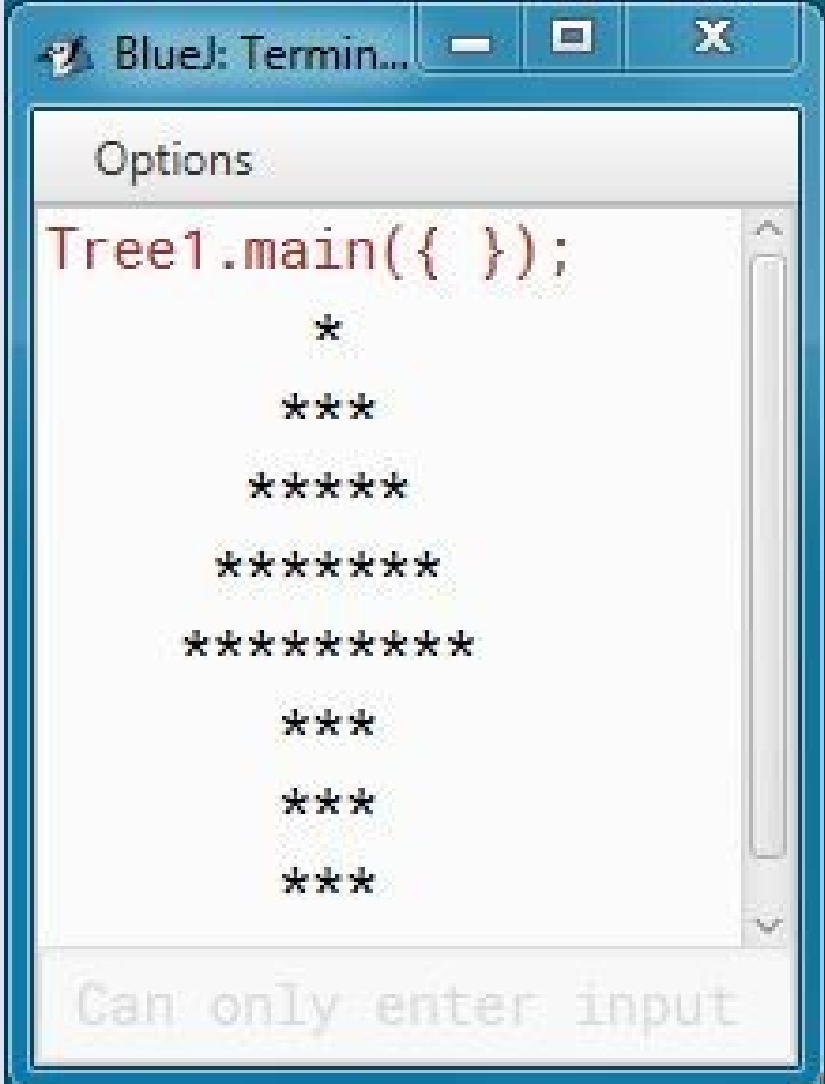
Below the code, the output of the program is shown as a tree structure made of star symbols. The tree has a height of 5, with the foliage (leaves) being 5 units high. The output is:

```
      *
     ***
    *****
   *********
  ***********
      ***
      ***
      ***
```

At the bottom of the window, there is a text input field with the placeholder text "Can only enter input".

An example – printing a tree

- We think first about printing just one tree, say this one.
- We notice that the tree is just a sequence of lines of characters, so we could just print each line.



```
BlueJ: Terminated
```

```
Options
```

```
Tree1.main({ });
```

```
    *
```

```
   ***
```

```
  *****
```

```
 *****
```

```
*****
```

```
  *****
```

```
   ***
```

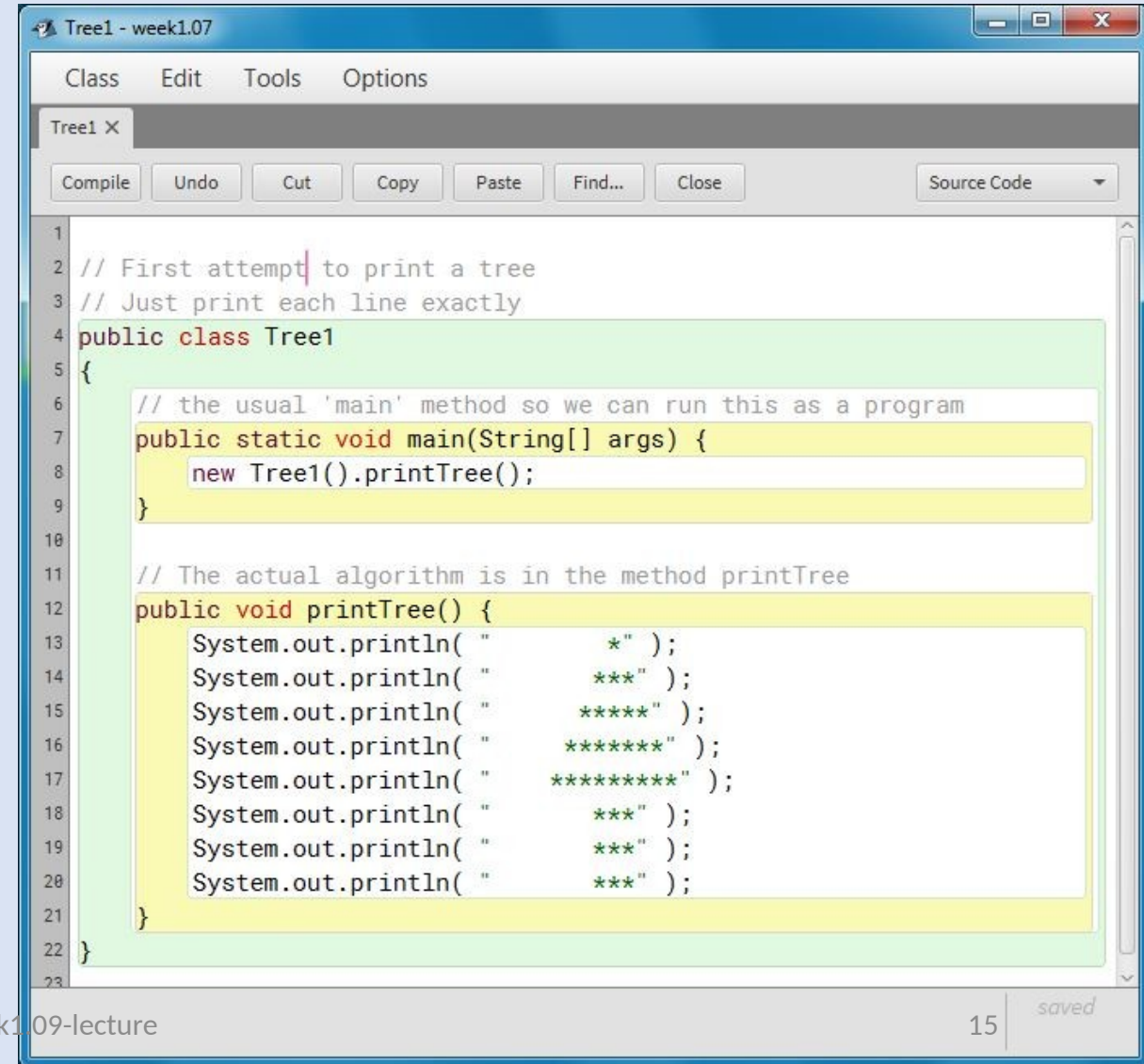
```
   ***
```

```
   ***
```

```
Can only enter input
```

An example – printing a tree

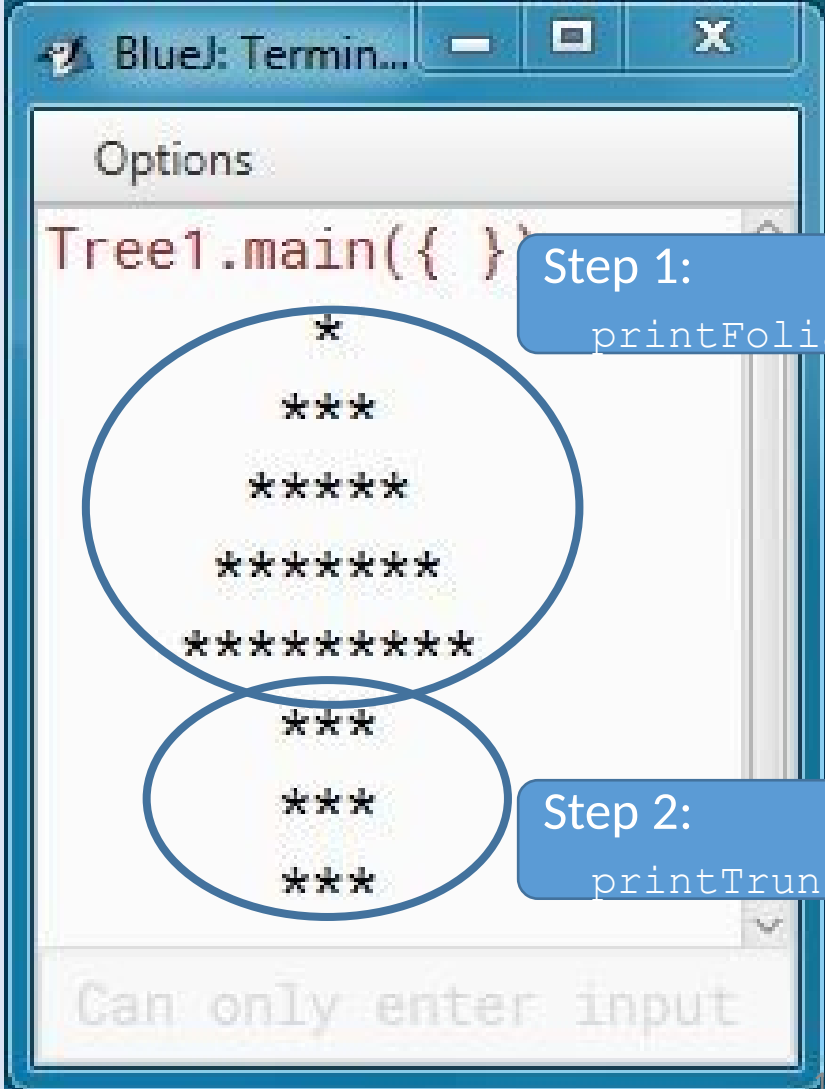
- Like this ([Tree1](#) in this week's BlueJ project)
- We get full marks for identifying a **sequence** in the task, but it only works for this one tree
- So we need to look a bit harder.



```
1 // First attempt to print a tree
2 // Just print each line exactly
3
4 public class Tree1
5 {
6     // the usual 'main' method so we can run this as a program
7     public static void main(String[] args) {
8         new Tree1().printTree();
9     }
10
11     // The actual algorithm is in the method printTree
12     public void printTree() {
13         System.out.println( "      *" );
14         System.out.println( "     ***" );
15         System.out.println( "    *****" );
16         System.out.println( "   *****" );
17         System.out.println( "  *****" );
18         System.out.println( " ***" );
19         System.out.println( " ***" );
20         System.out.println( " ***" );
21     }
22 }
```

An example – printing a tree

- Ok let's look harder – what do we notice?
- We're looking for two things really: ways of **breaking the task into parts**, and **patterns** within or between parts.
- There are two clear big **parts** here – the foliage and the trunk – and they are printed in **sequence**
- So we create methods for them called `printFoliage()` and `printTrunk()`



The screenshot shows a terminal window titled "BlueJ: Termin...". Inside, the code `Tree1.main({ })` has been executed, resulting in a tree pattern of asterisks. The pattern is divided into two sections by blue circles. The top section, labeled "Step 1: `printFoliage()`", contains five lines of asterisks: a single star, three stars, five stars, seven stars, and nine stars. The bottom section, labeled "Step 2: `printTrunk()`", contains three lines of three stars each. At the bottom of the terminal, a prompt says "Can only enter input".

```
Tree1.main({ })
```

```
  *
 ***
*****
*****
*****
```

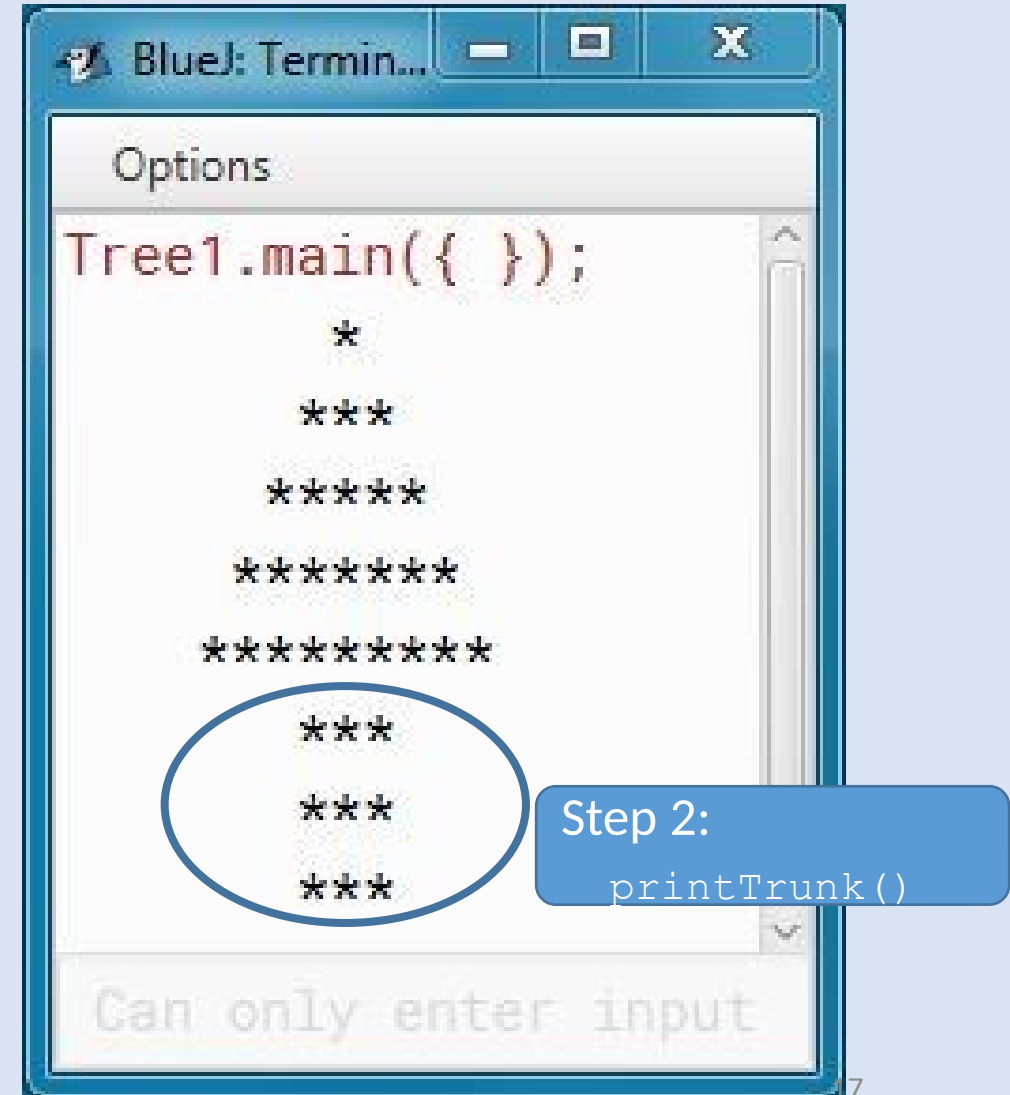
```
  ***
  ***
  ***
```

Can only enter input

An example – printing a tree

- Let's think about `printTrunk()` first, because it is easy.
- It just prints the same line 3 times.
- And it doesn't change even for different foliage
- So we can do this:

```
for (int i=0; i<3; i=i+1) {  
    System.out.println( "    ***" );  
}
```



An example – printing a tree

- Now `printFoliage()` which is harder – all its lines are different, and the number of them depends on the foliage height

```
Options
Tree1.main({ })

      *
     ***
    *****
   *********
  ***********
 *****
  ***
   ***
  ***

Can only enter input
```

Step 1:
`printFoliage()`

An example – printing a tree

- First the foliage pattern
- A good way to understand patterns is to construct a little table of what's changing on each line. For example, the number of stars printed is changing like this:

Line	numStars
0	1
1	3
2	5
3	7
4	9

```
Tree1.main({ })
```

```

      *
    ***
  *****
 *******
          *
        ***
       ***
      ***

```

Can only enter input

Step 1: printFoliage()

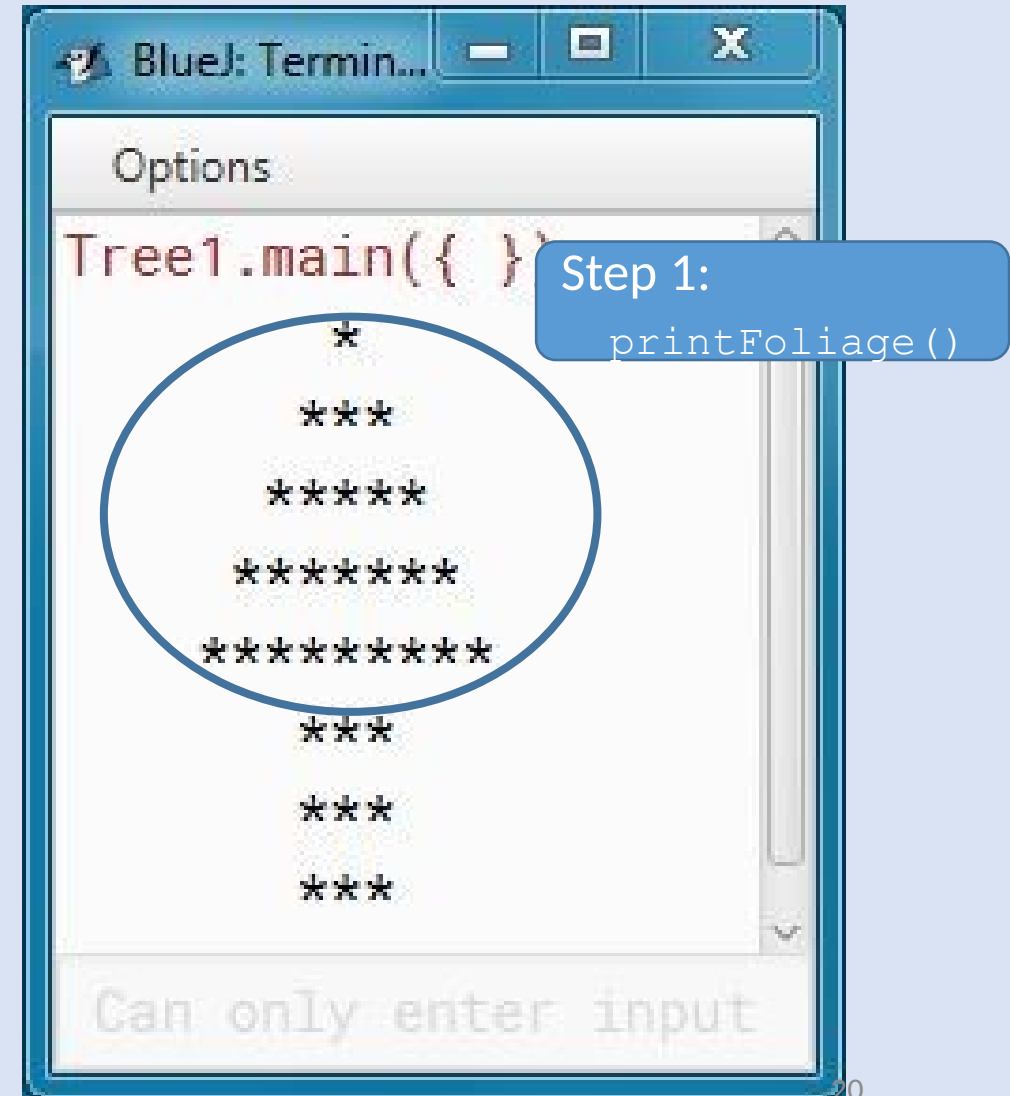
An example – printing a tree

- We want to turn this into a calculation – given the line number, what is the right number of stars?
- To do that, one good way is to look at how ‘stars’ changes from line to line:

Line	numStars	Change
0	1	
1	3	2
2	5	2
3	7	2
4	9	2

- This tells us that the calculation we want is

`numStars = 1 + line*2`



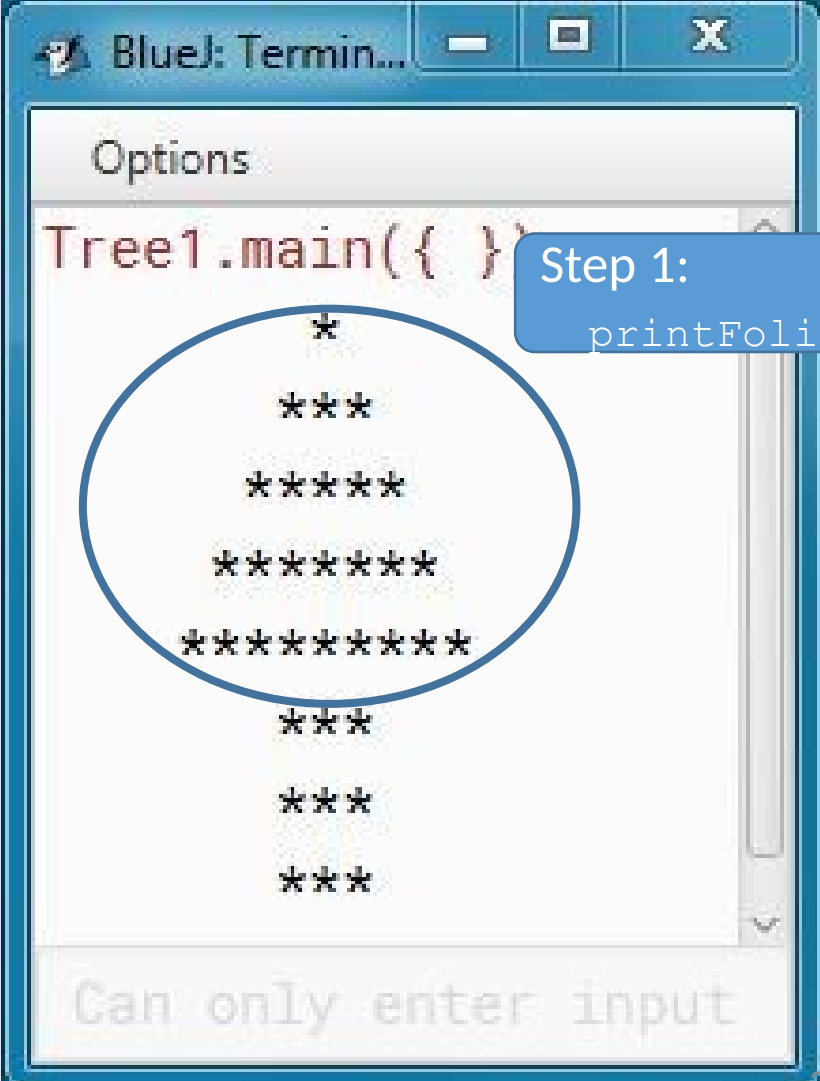
An example – printing a tree

- Now we turn our knowledge into loops:
 - One loop which prints each line
 - One loop inside that which prints the right number of stars

- So we can do this:

```
for (int line=0; line<5; line=line+1) {  
    int numStars = 1 + line*2;  
    for (int s=0; s < numStars; s=s+1) {  
        System.out.print("*" );  
    }  
    System.out.println();  
}
```

- And ta-dah! We print out ...



The screenshot shows a Java IDE window titled "BlueJ: Termin...". The main area displays the code `Tree1.main({ })` and a printed output of a tree pattern. The tree consists of five lines of stars: the first line has 1 star, the second has 3, the third has 5, the fourth has 7, and the fifth has 9. A blue oval highlights the first four lines of the tree. A blue callout box with the text "Step 1: printFoliage()" points to the first line of the tree. At the bottom of the IDE window, there is a text area with the placeholder text "Can only enter input".

```
Tree1.main({ })
```

```
      *  
     ***  
    *****  
   *********  
  ***********  
   ***  
    ***  
   ***
```

Can only enter input

An example – printing a tree

- Oops! That's not right!!! (but it's in `Tree2` anyway)
- We've got the stars right, but what about the spaces?
- Let's do our table thing again, for the spaces

Line	numSpaces	Change
0	8	
1	7	-1
2	6	-1
3	5	-1
4	4	-1

- This tells us that the calculation we want is

```
numSpaces = 8 + line*(-1);
```

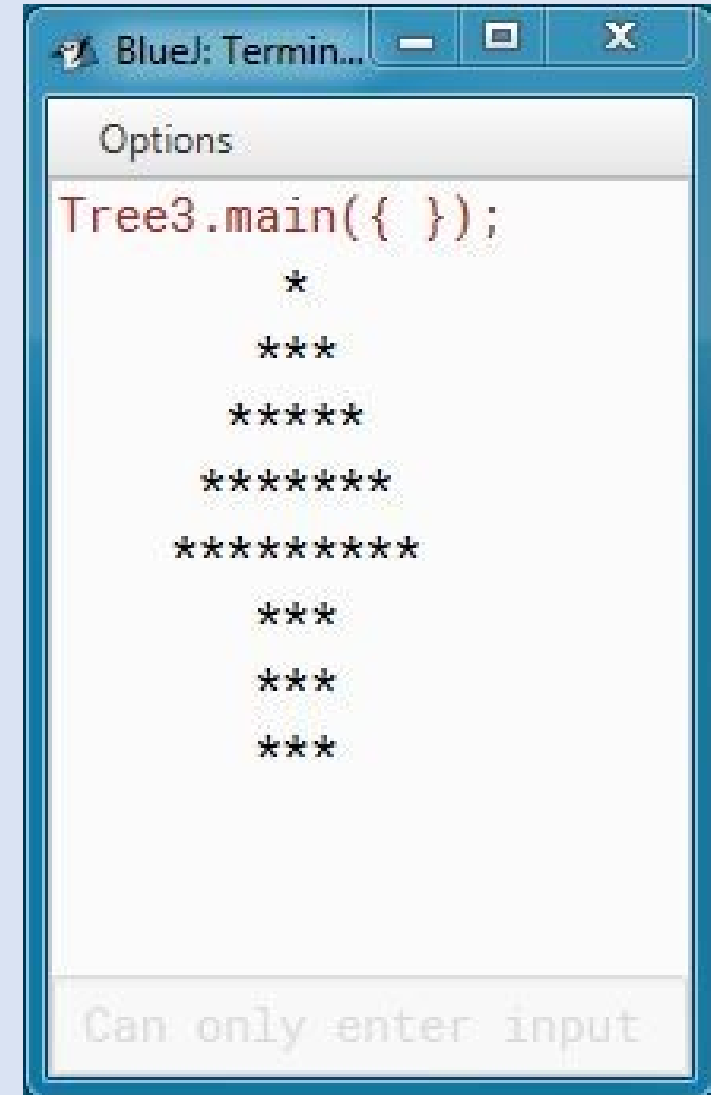
or

```
numSpaces = 8 - line;
```

```
BlueJ: Termin...  
Options  
Tree2.main({ });  
*  
***  
*****  
*****  
*****  
          ***  
          ***  
          ***  
  
Can only enter input
```

An example – printing a tree

- That's better! This is `Tree3`
- One more change to this version, before solving the last problem.
- In our solution to the foliage drawing, there are two more parts that we could identify and make into a sub-method (just for neatness) – writing multiple stars or spaces
- `Tree4` has the same code modified to include this new method, `printMany`.



```
BlueJ: Termin...  
Options  
Tree3.main({ });  
  
  *  
 ***  
*****  
*****  
*****  
      *  
      *  
      *
```

Can only enter input

Tree4

- Tree4, with the printMany method, which lets us print multiple spaces and stars.
- Now all we need to worry about is changing the height of the tree

```
9 // The actual algorithm is in the method printTree
10 // Here it just prints the foliage, and then the trunk, using two sub-methods
11 public void printTree() {
12     printFoliage();
13     printTrunk();
14 }
15
16 // Print the foliage lines
17 public void printFoliage() {
18     // outer loop runs the body for each line (assumes foliage has 5 lines)
19     for (int line=0; line<5; line=line+1) {
20         // on each line, work out how many spaces you need and print them
21         int numSpaces = 8 - line;
22         printMany(numSpaces, " ");
23         // now work out how many stars to print and print them
24         int numStars = 1 + line*2;
25         printMany(numStars, "*");
26         //remember to print a newline at the end of each line
27         System.out.println();
28     }
29 }
30
31 // printTrunk is easy, as it just prints three identical lines.
32 public void printTrunk() {
33     for (int i = 0; i < 3; i = i+1 ) {
34         System.out.println( "      ***" );
35     }
36 }
37
38 //printMany = print theString the number of times requested
39 public void printMany(int number, String theString) {
40     for (int s=0; s<number; s=s+1) {
41         System.out.print( theString );
42     }
43 }
44 }
45 }
```


An example – printing a tree

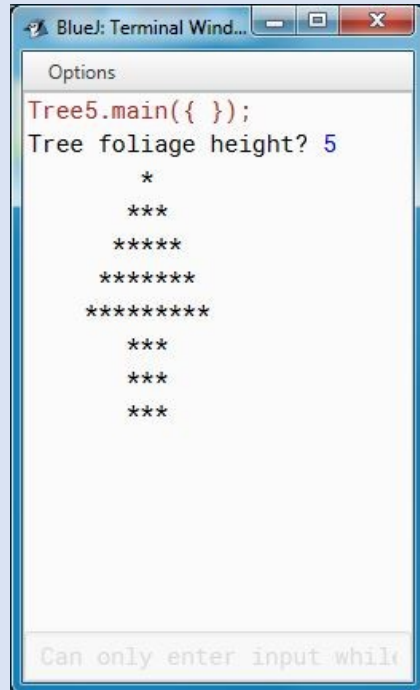
- We mentioned before that we use **variables** to control the operation of algorithms.
- This is a case of that – so far we have assumed a height of 5 and that is hardcoded into our code.
- We need to change it to a variable which the user can provide.
- **But does anything else need to change?**
- **Tree5** has this simple modification, with a variable **height** read in from the user.
- Let's test it with a few values

Tree5

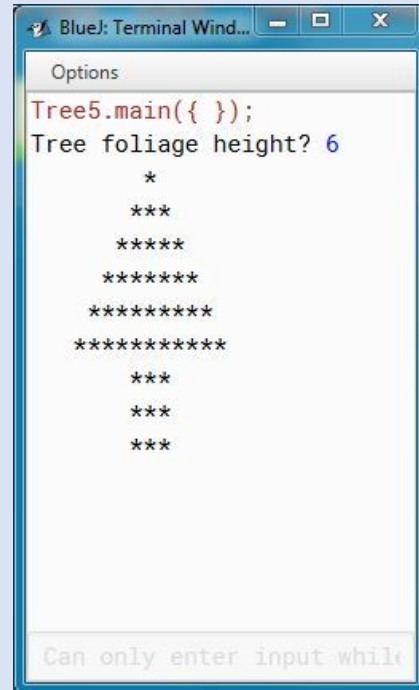
- Tree5, with a height variable which is read in using a Scanner object, and then passed to printFoliage

```
11 // The actual algorithm is in the method printTree
12 // Here it ask the user to specify the foliage height, and then
13 // prints the foliage, and then the trunk, using two sub-methods
14 public void printTree() {
15     Scanner myScanner = new Scanner(System.in);
16     System.out.print("Tree foliage height? ");
17     int height = myScanner.nextInt();
18     printFoliage(height);
19     printTrunk();
20     myScanner.close();
21 }
22
23 // Print the foliage lines
24 public void printFoliage(int height) {
25     // outer loop runs the body for each line (assumes foliage has 5 lines)
26     for (int line=0; line<height; line=line+1) {
27         // on each line, work out how many spaces you need and print them
28         int numSpaces = 8 - line;
29         printMany(numSpaces, " ");
30         // now work out how many stars to print and print them
31         int numStars = 1 + line*2;
32         printMany(numStars, "*");
33         //remember to print a newline at the end of each line
34         System.out.println();
35     }
36 }
```

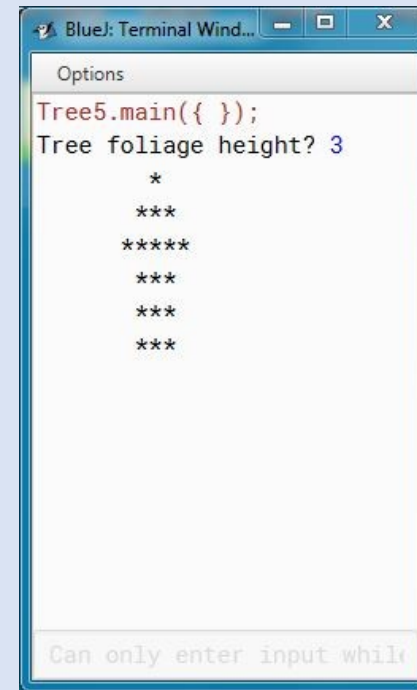
Testing Tree5



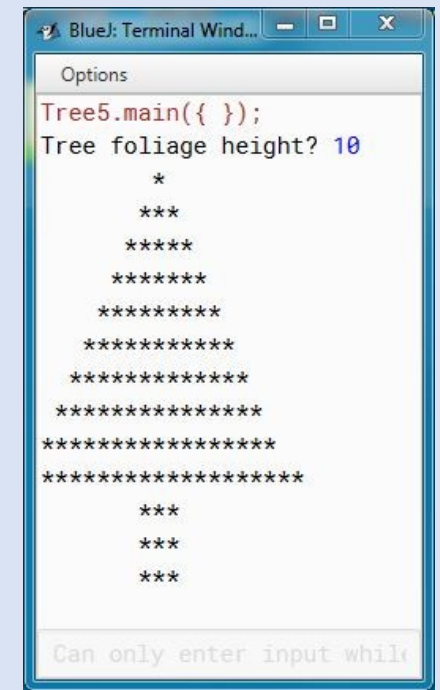
```
BlueJ: Terminal Wind...
Options
Tree5.main({ });
Tree foliage height? 5
  *
 ***
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree5.main({ });
Tree foliage height? 6
  *
 ***
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree5.main({ });
Tree foliage height? 3
  *
 ***
*****
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree5.main({ });
Tree foliage height? 10
  *
 ***
*****
*****
*****
*****
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```

An example – printing a tree

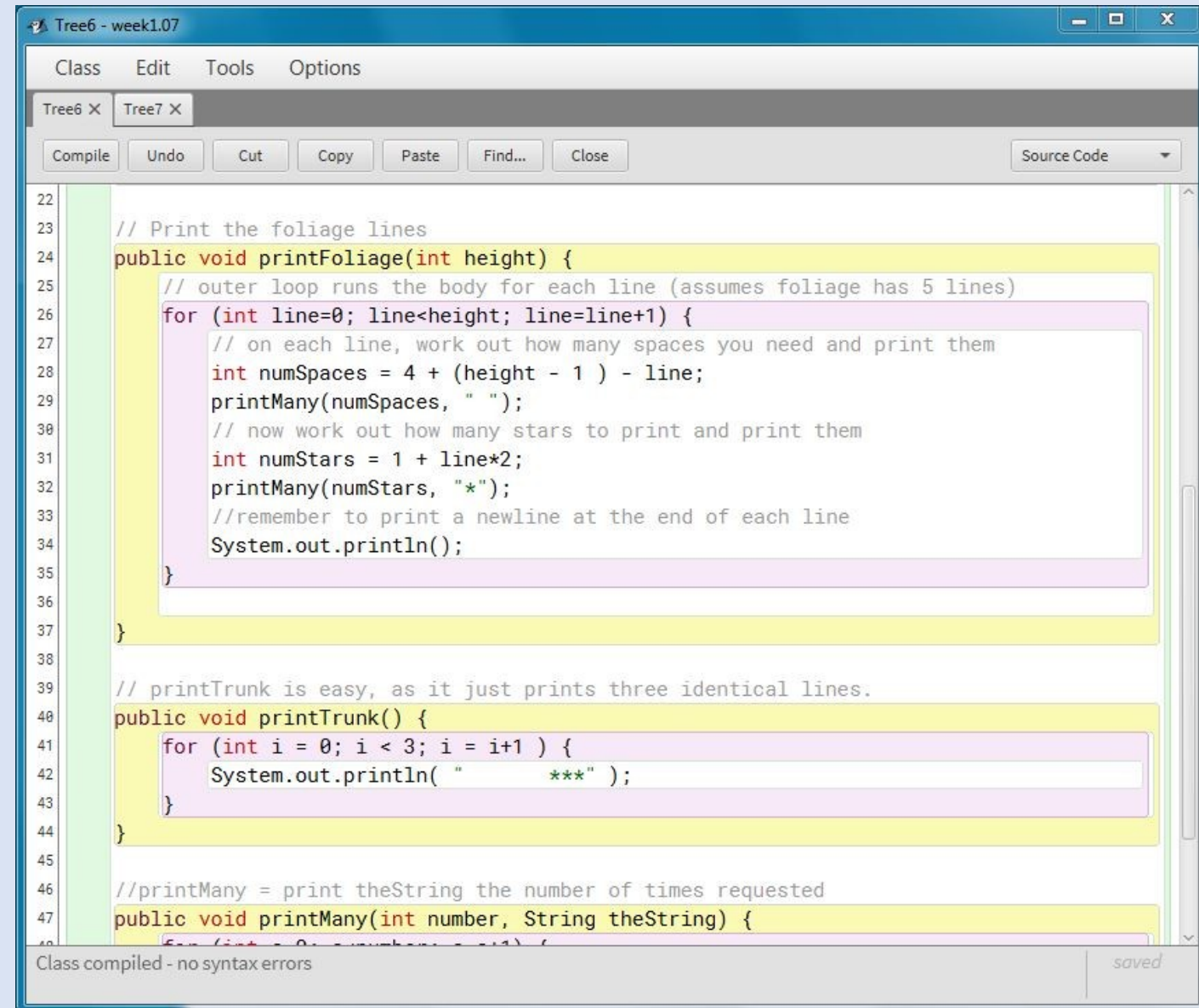
- The last example shows there is still something wrong.
- In fact there's something wrong with all of them except the original one.
- The algorithm is not keeping the left margin to be 4 spaces.
- So when the tree gets too wide, its shape breaks.

An example – printing a tree

- 5 was **not** the only hardcoded number in the old algorithm
- The number of spaces was calculated as `8-line`, but the value 8 depends on the number of stars in the widest part of the tree, and that depends on its height.
- The widest row is number `height-1` so the number of stars on it is $1 + (\text{height}-1) * 2$. The number of stars on row 0 is 1, and the number of extra spaces row 0 needs (in addition to the 4 margin spaces) is half the difference between the two which is $((1 + (\text{height}-1) * 2) - 1) / 2$ which is `height-1`.
- So 8 should be $4 + (\text{height}-1)$ which just happens to be 8 when height 5
- Let's look at our output again, with this new calculation

Tree6

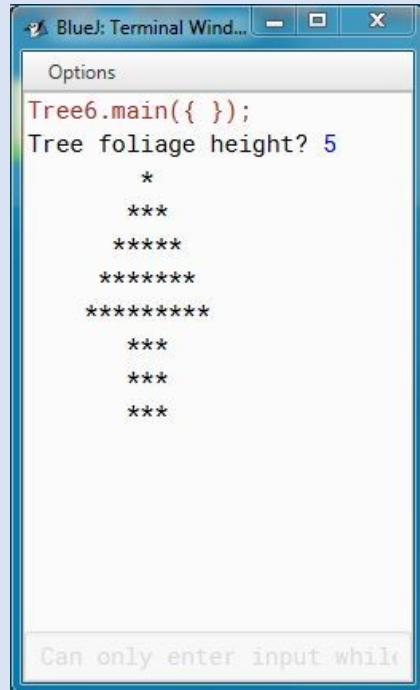
- Tree6, with corrected calculation of the number of spaces



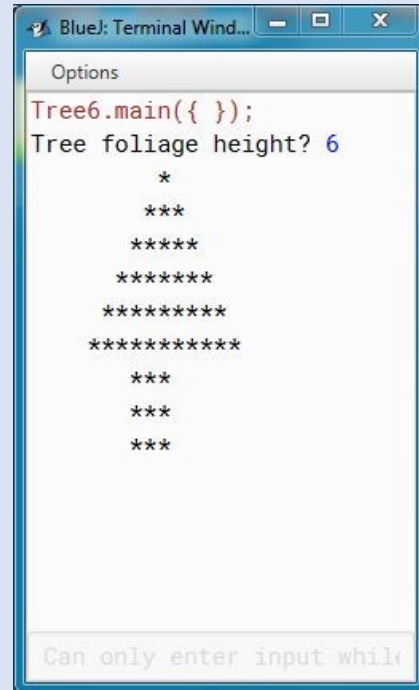
```
22
23 // Print the foliage lines
24 public void printFoliage(int height) {
25     // outer loop runs the body for each line (assumes foliage has 5 lines)
26     for (int line=0; line<height; line=line+1) {
27         // on each line, work out how many spaces you need and print them
28         int numSpaces = 4 + (height - 1 ) - line;
29         printMany(numSpaces, " ");
30         // now work out how many stars to print and print them
31         int numStars = 1 + line*2;
32         printMany(numStars, "*");
33         //remember to print a newline at the end of each line
34         System.out.println();
35     }
36 }
37
38
39 // printTrunk is easy, as it just prints three identical lines.
40 public void printTrunk() {
41     for (int i = 0; i < 3; i = i+1 ) {
42         System.out.println( "      ***" );
43     }
44 }
45
46 //printMany = print theString the number of times requested
47 public void printMany(int number, String theString) {
48     for (int i = 0; i < number; i = i+1) {
49         System.out.print(theString);
50     }
51 }
```

Class compiled - no syntax errors

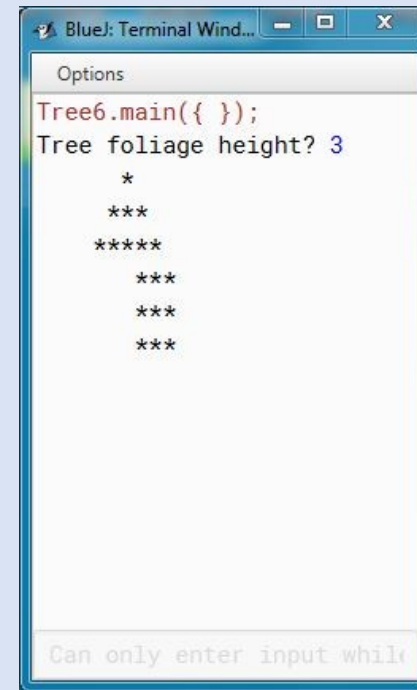
Testing Tree6



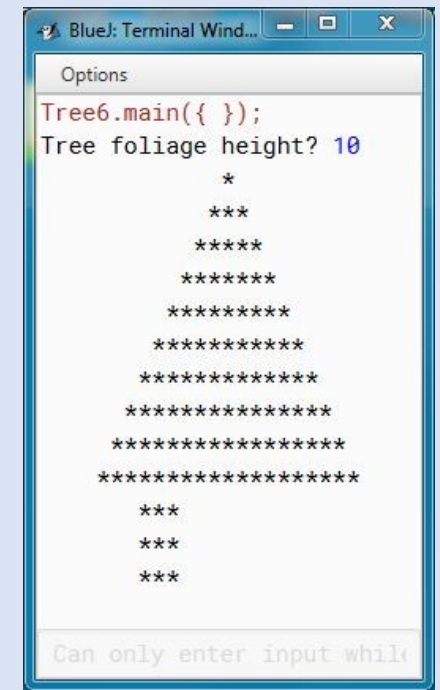
```
BlueJ: Terminal Wind...
Options
Tree6.main({ });
Tree foliage height? 5
  *
 ***
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree6.main({ });
Tree foliage height? 6
  *
 ***
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree6.main({ });
Tree foliage height? 3
  *
 ***
*****
  ***
  ***
Can only enter input while
```



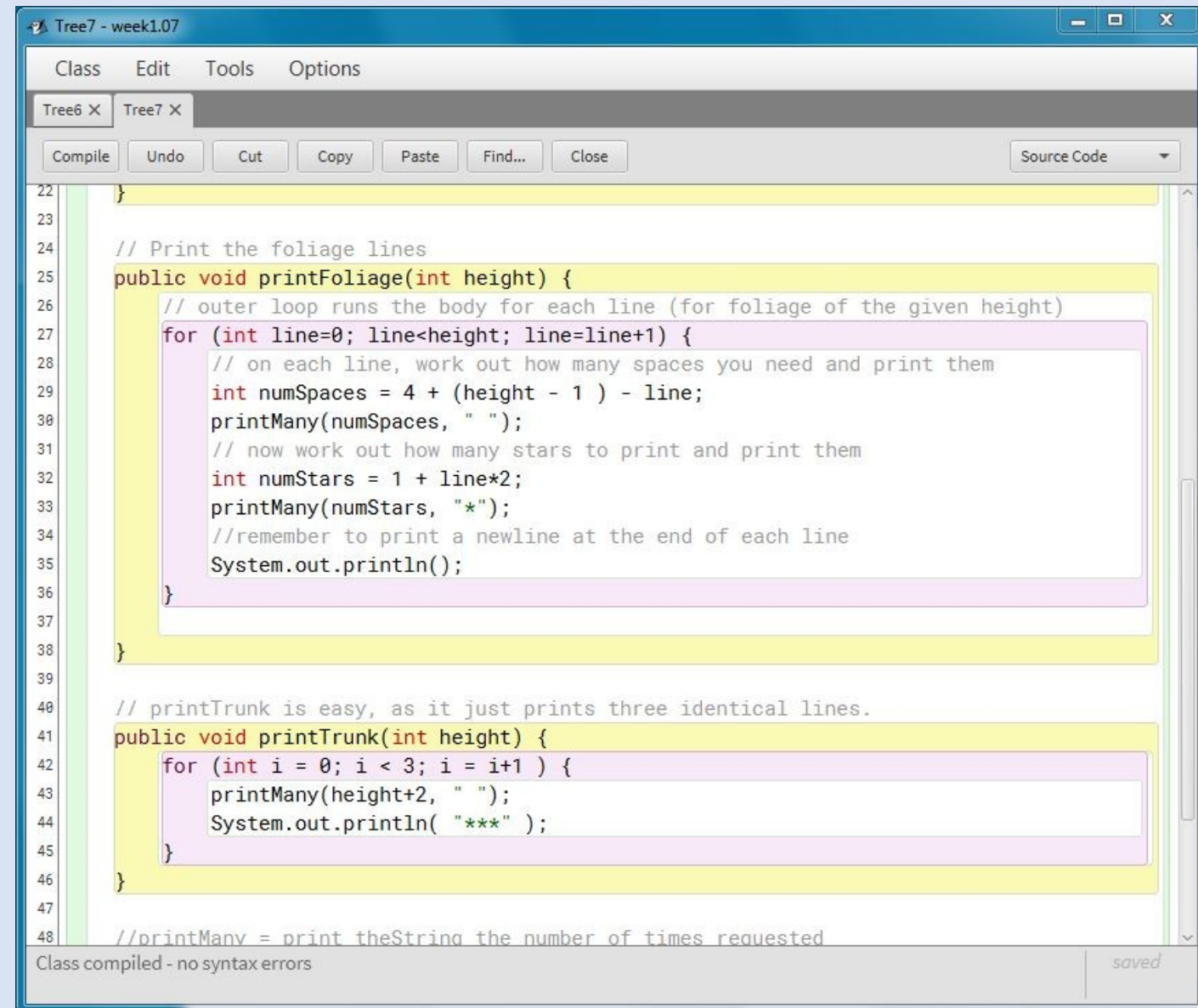
```
BlueJ: Terminal Wind...
Options
Tree6.main({ });
Tree foliage height? 10
  *
 ***
*****
*****
*****
*****
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```

An example – printing a tree

- We are still not quite there.
- It seems the trunk component was not as straightforward as we thought – it also needs to vary the number of spaces according to the height.
- It's quite easy to work out how many spaces it needs though – it is exactly the same as line 1 of the foliage, so the number of spaces it needs is $4 + (\text{height} - 1) - 1$ which is $\text{height} + 2$

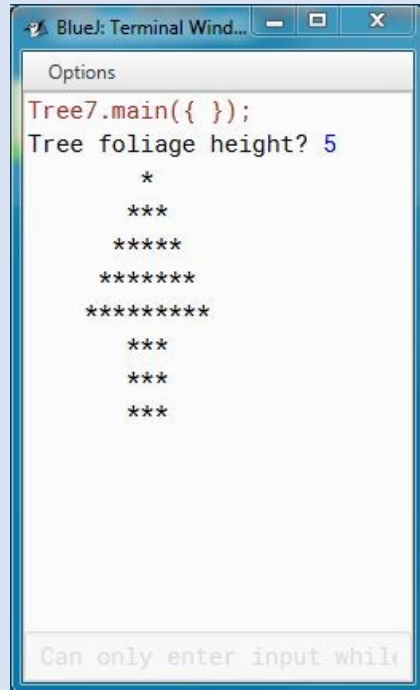
Tree7

- Tree7, with corrected calculation of the number of spaces for trunk also.
- This version works!

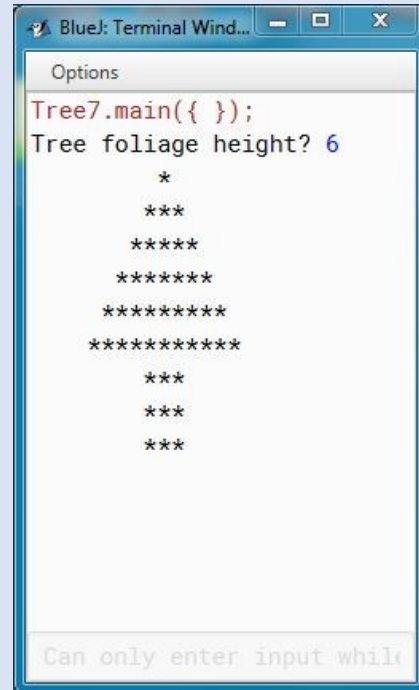


```
Tree7 - week1.07
Class Edit Tools Options
Tree6 X Tree7 X
Compile Undo Cut Copy Paste Find... Close Source Code
22 }
23
24 // Print the foliage lines
25 public void printFoliage(int height) {
26     // outer loop runs the body for each line (for foliage of the given height)
27     for (int line=0; line<height; line=line+1) {
28         // on each line, work out how many spaces you need and print them
29         int numSpaces = 4 + (height - 1) - line;
30         printMany(numSpaces, " ");
31         // now work out how many stars to print and print them
32         int numStars = 1 + line*2;
33         printMany(numStars, "*");
34         //remember to print a newline at the end of each line
35         System.out.println();
36     }
37 }
38
39
40 // printTrunk is easy, as it just prints three identical lines.
41 public void printTrunk(int height) {
42     for (int i = 0; i < 3; i = i+1) {
43         printMany(height+2, " ");
44         System.out.println( "***" );
45     }
46 }
47
48 //printMany = print theString the number of times requested
Class compiled - no syntax errors saved
```

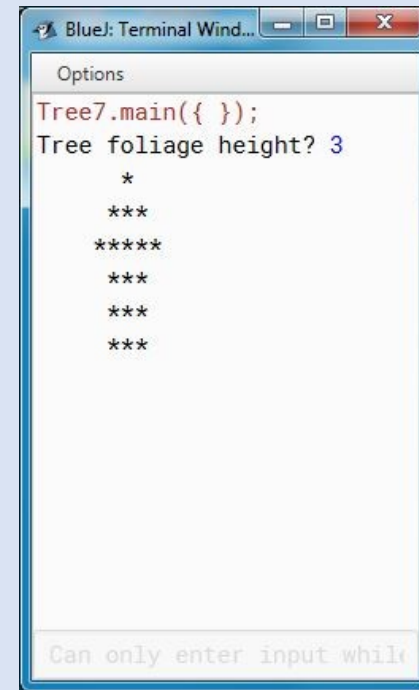
Testing Tree7



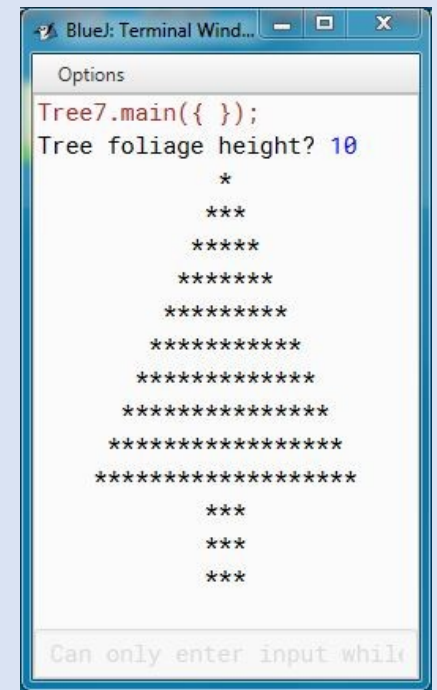
```
BlueJ: Terminal Wind...
Options
Tree7.main({ });
Tree foliage height? 5
  *
 ***
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree7.main({ });
Tree foliage height? 6
  *
 ***
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree7.main({ });
Tree foliage height? 3
  *
 ***
*****
  ***
  ***
Can only enter input while
```



```
BlueJ: Terminal Wind...
Options
Tree7.main({ });
Tree foliage height? 10
  *
 ***
*****
*****
*****
*****
*****
*****
*****
*****
  ***
  ***
  ***
Can only enter input while
```

Summary

- Well, it took a while but we got there!
- We saw how we have to look for **sequences**, **parts** and **patterns**, how to work out how to code loops by looking for **numerical patterns**, and how it is useful to break the task down into parts using **additional methods**.
- But we also saw the danger of working with just one example to get things going. It can be quite difficult to correctly fix some problems when you try and generalise.
- In the labs you will have some more examples to try yourself.

Lab exercises

Week 1.09

Lab exercises – Week 1.09

- This week's BlueJ project includes the Tree example (all seven attempts!) and three Lab files:
 - Lab1 gives you practice at constructing algorithms for simple shapes
 - Lab2 suggests some more complex shapes to try (nb: circle is quite hard, so don't worry if you can't do it!)
 - Lab3 is more advanced – it asks you to create an easier way of drawing by using a buffer array