Recursion

- Recursive Thinking
- Recursive Programming
- Recursion versus Iteration
- Direct versus Indirect Recursion
- More on Project 2
- Reading L&C 7.1 – 7.2

Recursive Thinking

- Many common problems can be stated in terms of a “base case” and an “inferred sequence of steps” to develop all examples of the problem statement from the base case
- Let’s look at one possible definition of a comma separated values (.csv) list:
  - A list can contain one item (the base case)
  - A list can contain one item, a comma, and a list (the inferred sequence of steps)

Recursive Thinking

- The above definition of a list is recursive because the second portion of the definition depends on there already being a definition for a list
- The second portion sounds like a circular definition that is not useful, but it is useful as long as there is a defined base case
- The base case gives us a mechanism for ending the circular action of the second portion of the definition

Recursive Thinking

- Using the recursive definition of a list:

  A list is a: number

  A list is a: number comma list

- Leads us to conclude 24, 88, 40, 37 is a list

```
number comma list
  24     ,    88, 40, 37
              number comma list
                88     ,    40, 37
                            number comma list
                              40     ,    37
                                          number
                                            37
```

Recursive Thinking

- Note that we keep applying the recursive second portion of the definition until we reach a situation that meets the first portion of the definition (the base case)
- Then we apply the base case definition
- What would have happened if we did not have a base case defined?

Infinite Recursion

- If there is no base case, use of a recursive definition becomes infinitely long and any program based on that recursive definition will never terminate and produce a result
- This is similar to having an inappropriate or no condition statement to end a “for”, “while”, or “do … while” loop

Recursion in Math

- One of the most obvious math definitions that can be stated in a recursive manner is the definition of integer factorial
- The factorial of a positive integer N (N!) is defined as the product of all integers from 1 to the integer N (inclusive)
- That definition can be restated recursively

$1! = 1$	(the base case)
$N! = N * (N - 1)!$	(the recursion)

Recursion in Math

- Using that recursive definition to get 5!

  5! = 5 * (5-1)!

  5! = 5 * 4 * (4-1)!

  5! = 5 * 4 * 3 * (3-1)!

  5! = 5 * 4 * 3 * 2 * (2-1)!

  5! = 5 * 4 * 3 * 2 * 1! (the base case)

  5! = 5 * 4 * 3 * 2 * 1

  5! = 120

Recursive Programming

- Recursive programming is an alternative way to program loops without using “for”, “while”, or “do … while” statements
- A Java method can call itself
- A method that calls itself must choose to continue using either the recursive definition or the base case definition
- The sequence of recursive calls must make progress toward meeting the definition of the base case

Recursion versus Iteration

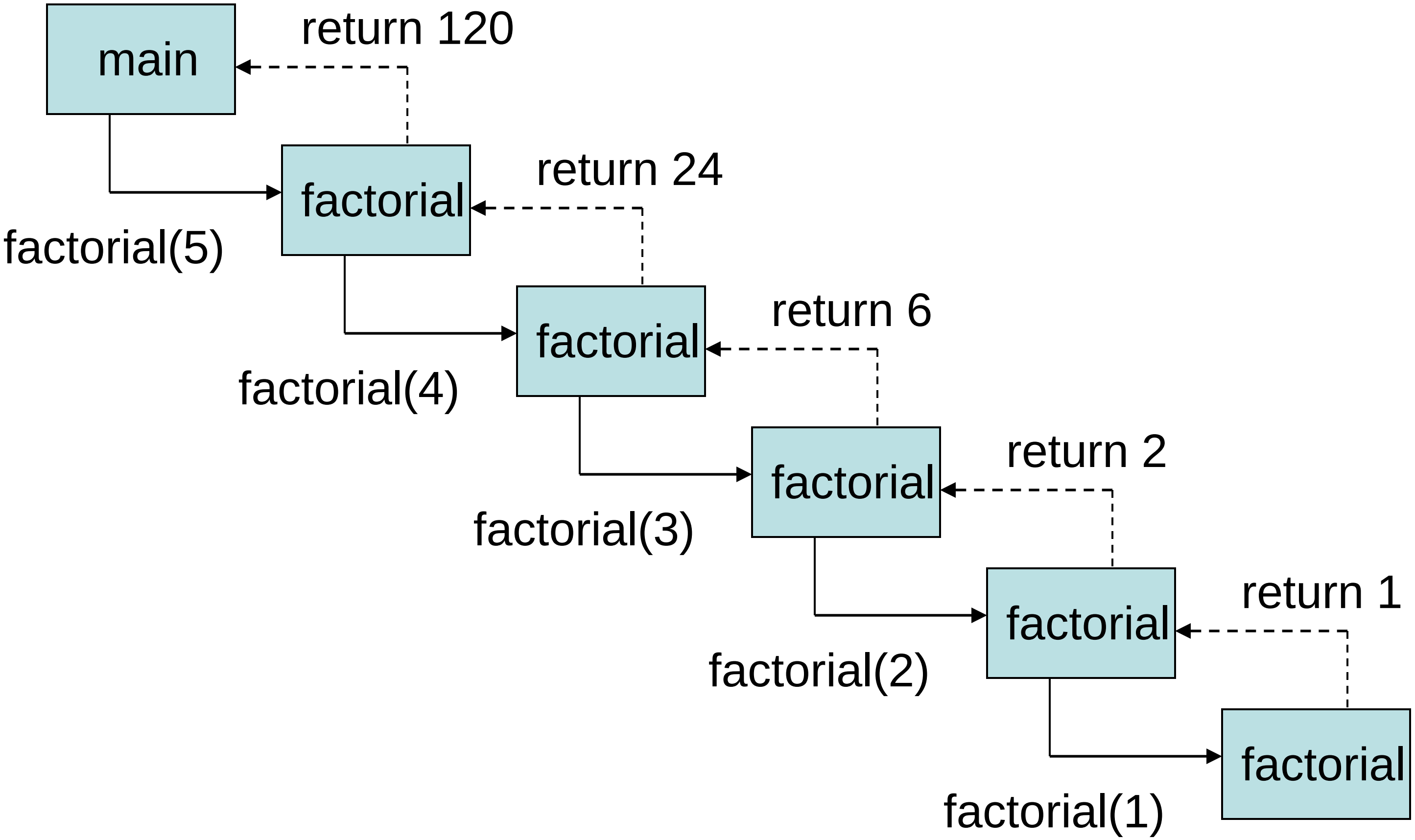
- We can calculate 5! using a loop

```
int fiveFactorial = 1;
for (int i = 1; i <= 5; i++)
   fiveFactorial *= i;
```

- Or we can calculate 5! using recursion

```
int fiveFactorial = factorial(5);
. . .
private int factorial(int n)
{
  return n == 1? 1 : n * factorial(n - 1);
}
```

Recursion versus Iteration



Recursion versus Iteration

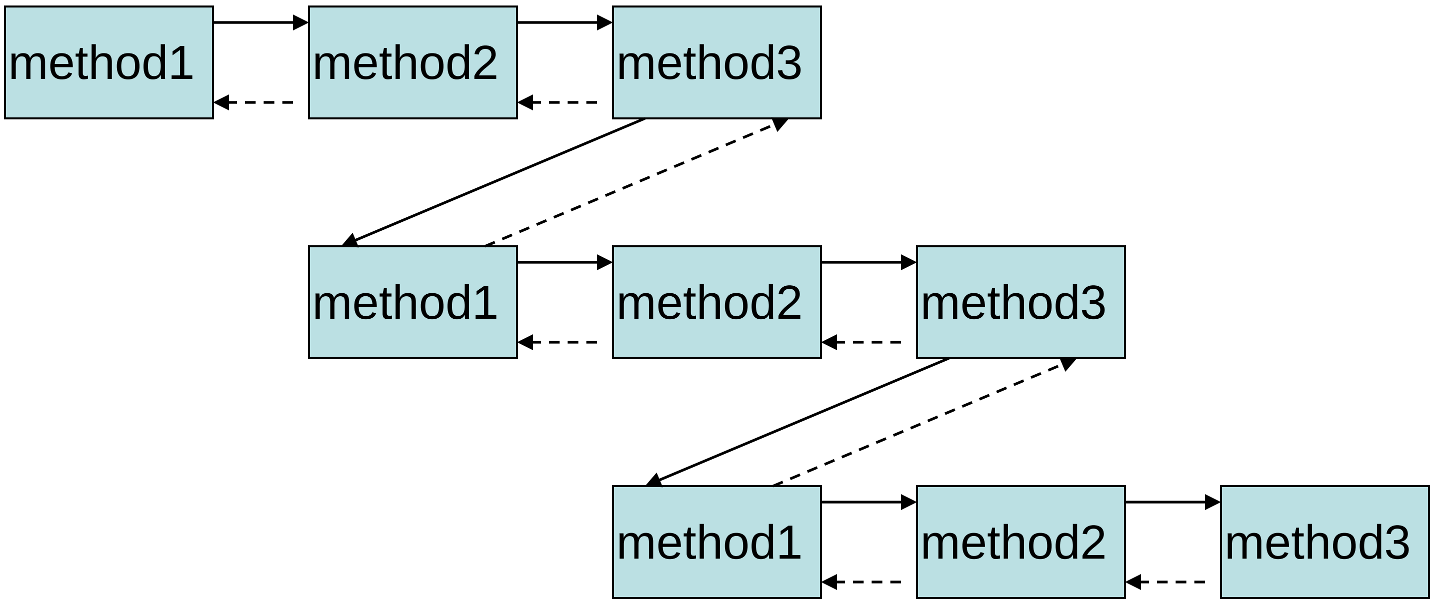
- Note that in the “for” loop calculation, there is only one variable containing the factorial value in the process of being calculated
- In the recursive calculation, a new variable n is created on the system stack each time the method factorial calls itself
- As factorial calls itself proceeding toward the base case, it pushes the current value of n-1
- As factorial returns after the base case, the system pops the now irrelevant value of n-1

Recursion versus Iteration

- Note that in the “for” loop calculation, there is only one addition (i++) and a comparison (i<=5) needed to complete each loop
- In the recursive calculation, there is a comparison (n==1) and a subtraction (n - 1), but there is also a method call/return needed to complete each loop
- Typically, a recursive solution uses both more memory and more processing time than an iterative solution

Direct versus Indirect Recursion

- Direct recursion is when a method calls itself
- Indirect recursion is when a method calls a second method (and/or perhaps subsequent methods) that can call the first method again



Calling main( ) Recursively ☺

- Any Java method can call itself
- Even `main()` can call itself as long as there is a base case to end the recursion
- You are restricted to using a `String []` as the parameter list for `main()`
- The JVM requires the main method of a class to have that specific parameter list

Calling main( ) Recursively ☺

```
public class RecursiveMain
{
  public static void main(String[] args)
  {
    if (args.length > 1)   {
      String [] newargs = new String[args.length - 1];
      for (int i = 0; i < newargs.length; i++)
        newargs[i] = args[i + 1];
      main(newargs);                    // main calls itself with a new args array
    }
    System.out.println(args[0]);
    return;
  }
}
```

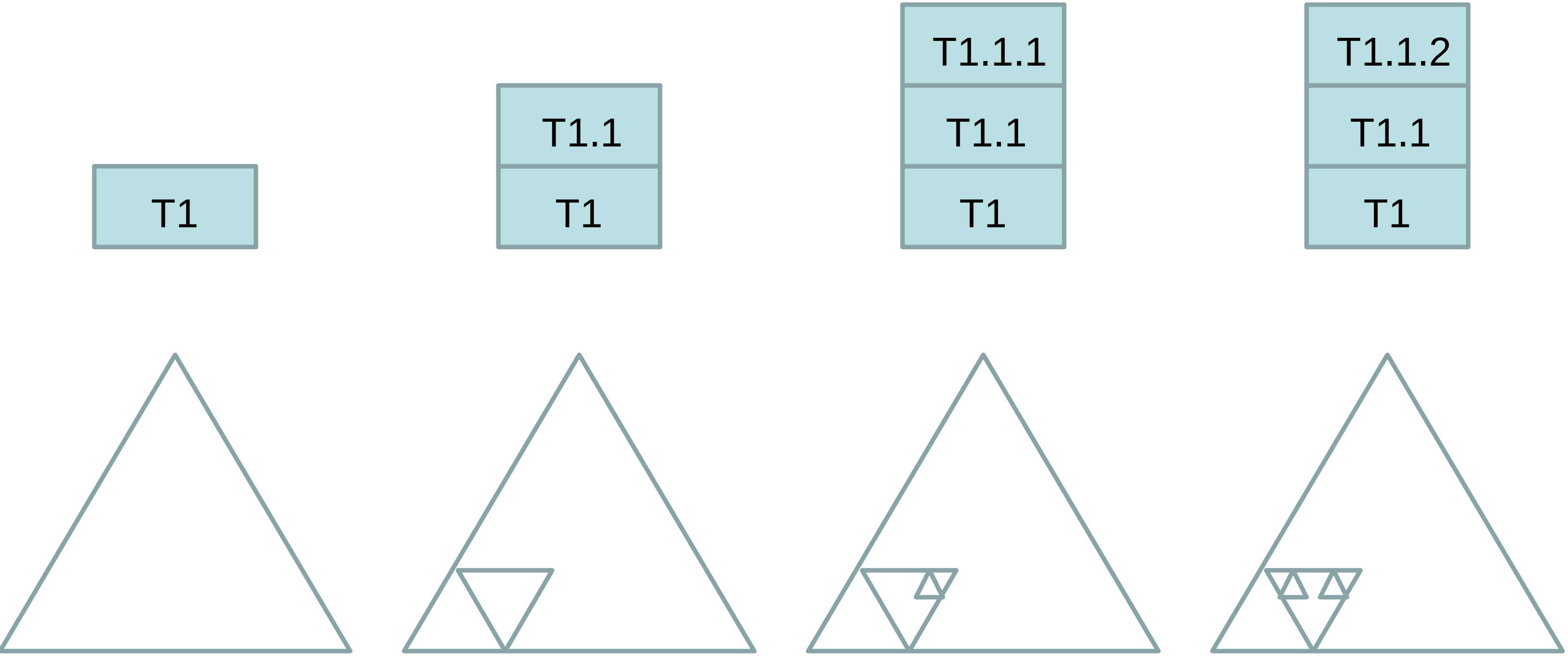
```
java RecursiveMain computer science is fun
fun
is
science
computer
```

More on Project 2

- The Recursive class for project 2 needs a recursive `drawTriangle()` method
- You don’t need an explicit stack
- When `drawTriangle()` calls itself:
  – the current context of all local variables is left on the system stack and
  – a new context for all local variables is created on the top of the system stack
- The return from `drawTriangle()` pops the previous context off the system stack

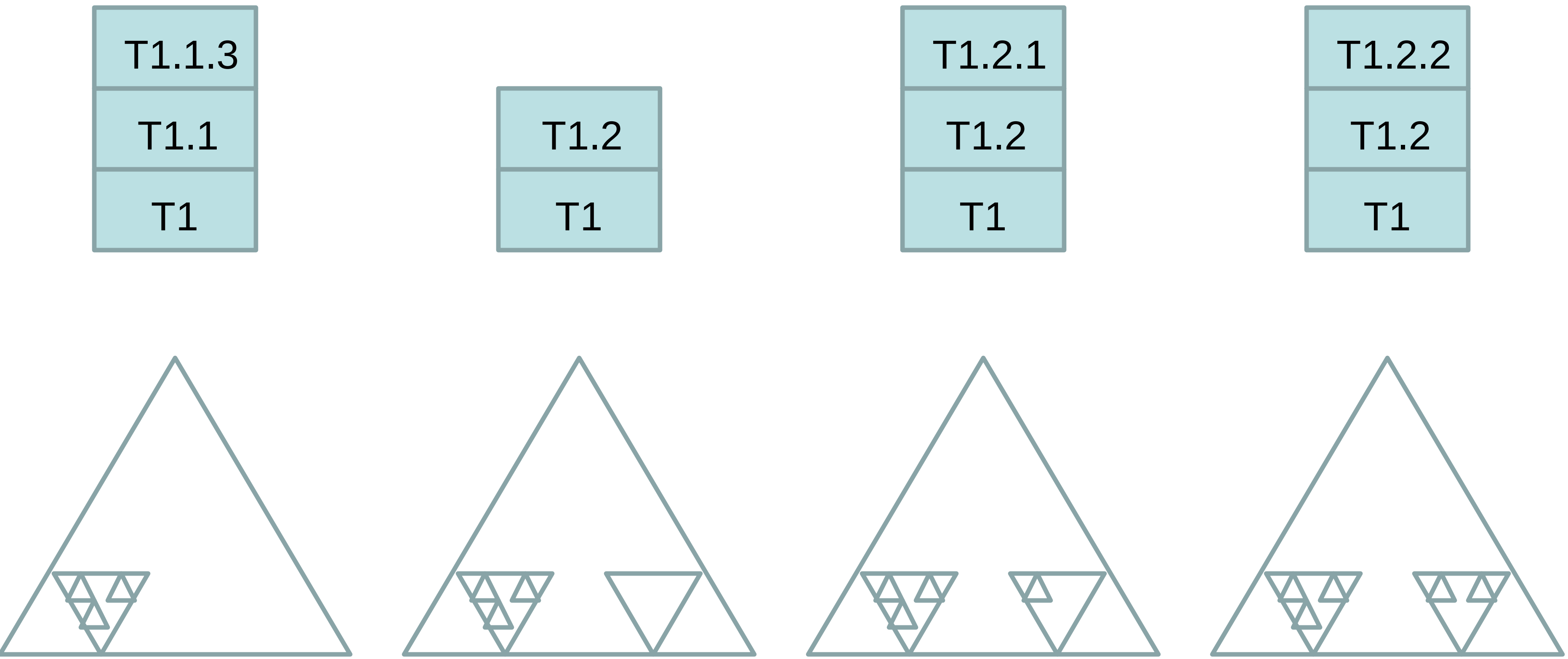
More on Project 2

- System Stack (for 3 triangles and 3 levels)



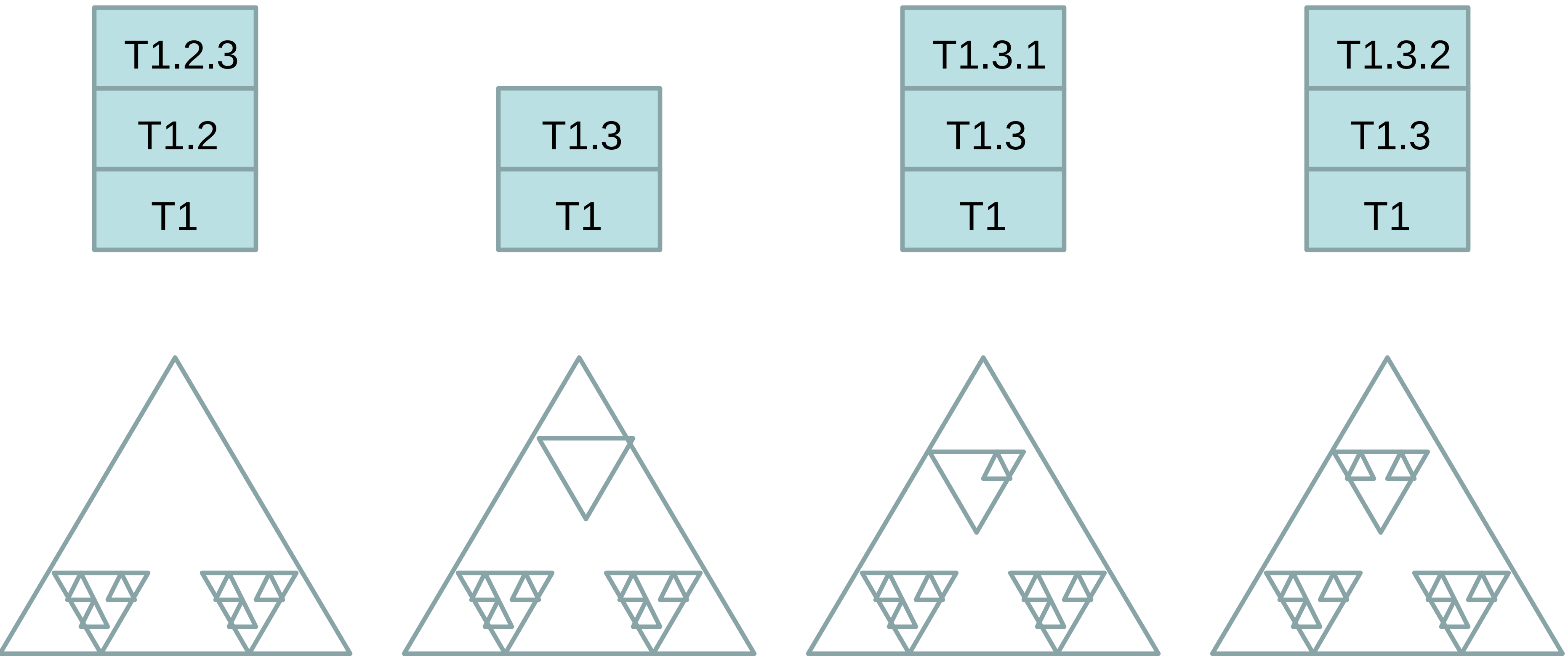
More on Project 2

- System Stack (for 3 triangles and 3 levels)



More on Project 2

- System Stack (for 3 triangles and 3 levels)



More on Project 2

- System Stack (for 3 triangles and 3 levels)