Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Online Course

```java
class Lecture6 {

                    "Methods"

}

// Keywords:
return
```

Methods[1]

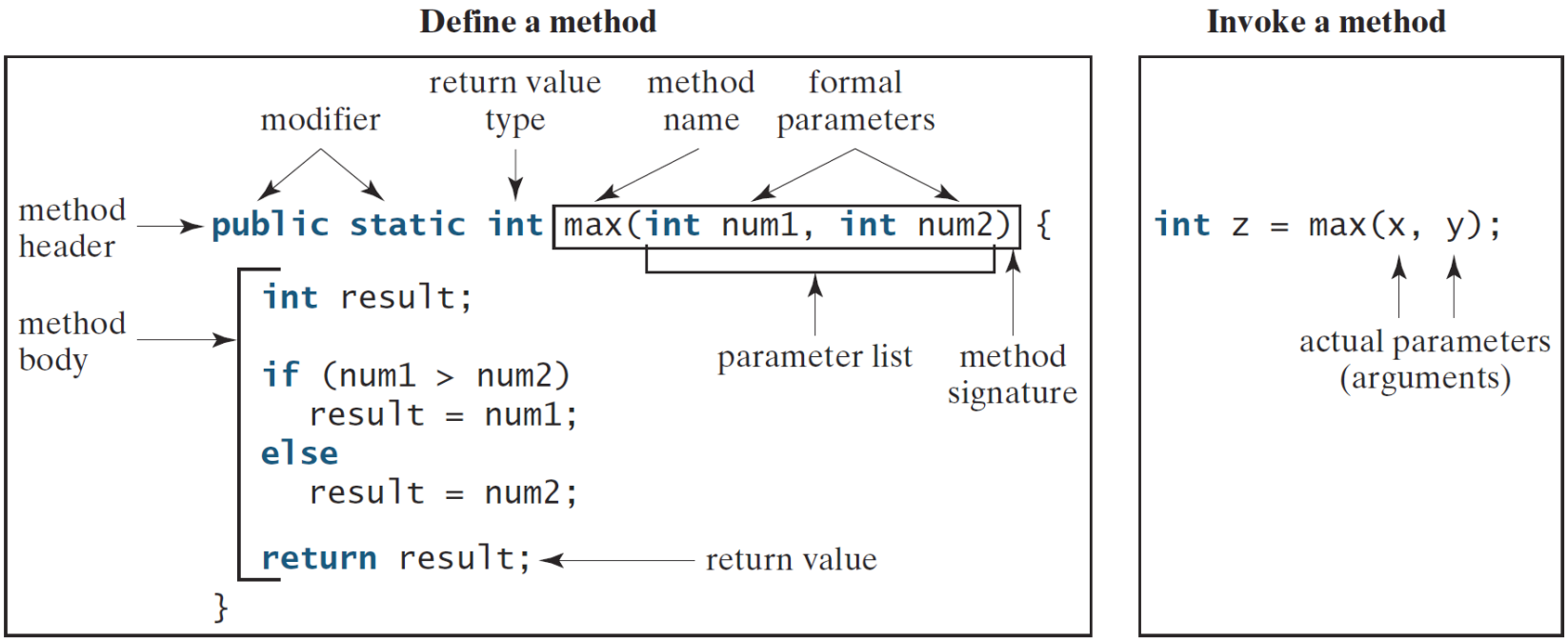
- Methods (or functions) can be used to define reusable code, and so that it could organize and simplify code.
- The idea of methods/functions originates from math, like

$$f(x, y),$$

  where x and y denote two input parameters.
- In computer science, each input parameter should be declared with a specific type.
- Moreover, a function should be assigned with a return type!

[1] Aka procedures and subroutine.

Example: max



- The method name and the parameter list together are called the method signature.[2]

[2] Method overloading depends signatures. We will see it soon.

Alternatives?

```
1  ...
2      public static int max(int num1, int num2) {
3
4          if (num1 > num2) {
5              return num1;
6          } else {
7              return num2;
8          }
9
10     }
11 ...
```

```
1  ...
2      public static int max(int num1, int num2) {
3
4          return num1 > num2 ? num1 : num2;
5
6      }
7  ...
```

"All roads lead to Rome."

– Anonymous

"但如你根本並無招式，敵人如何來破你的招式？"

– 風清揚，笑傲江湖。第十回。傳劍

About return

- The return statement is used to end the method.
- We say that a callee is the method invoked by a caller.
- The caller has obligation to provide inputs to the callee and expect the returned value.
- The callee should guarantee to return a value.
- This establishes the relation (right/obligation) between both.
- Once one specifies the return type (except void), this method should guarantee to return a value of that type.

Pitfalls

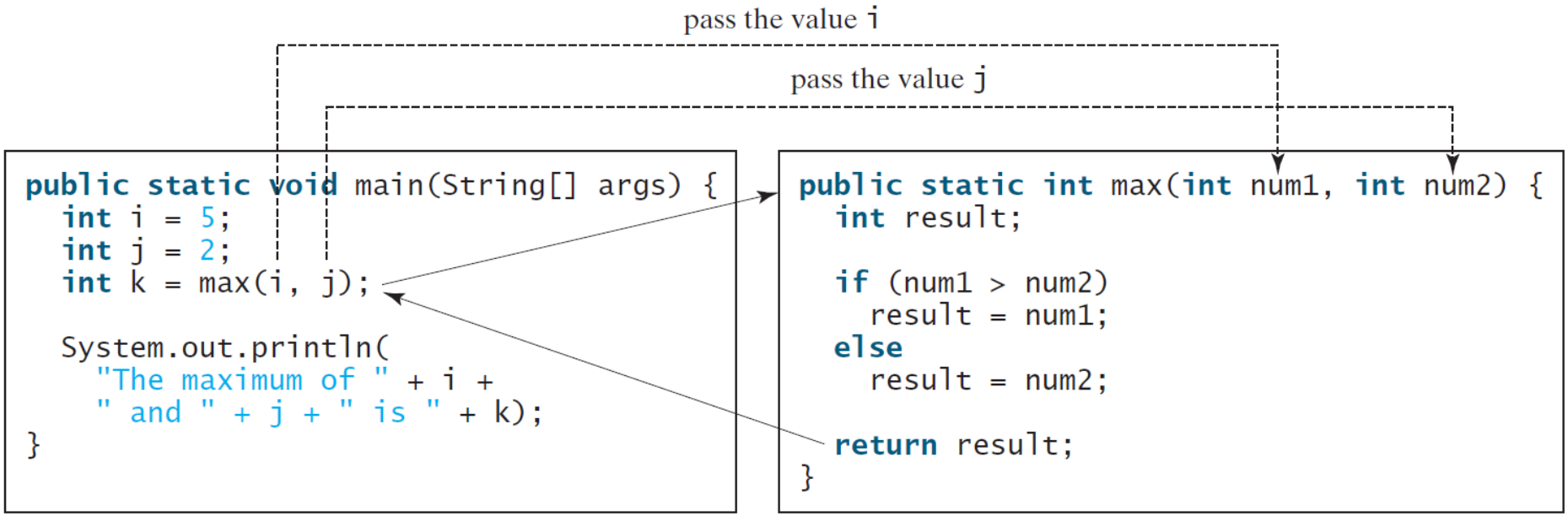
- The following two methods are incorrect.

```
1  ...
2      public static int foo1() {
3
4          while (true);
5          return 0; // Unreachable code.
6
7      }
8
9      public static int foo2(int x) {
10
11         if (x > 0)
12             return x; // What if x <= 0?
13
14     }
15 ...
```

More Examples

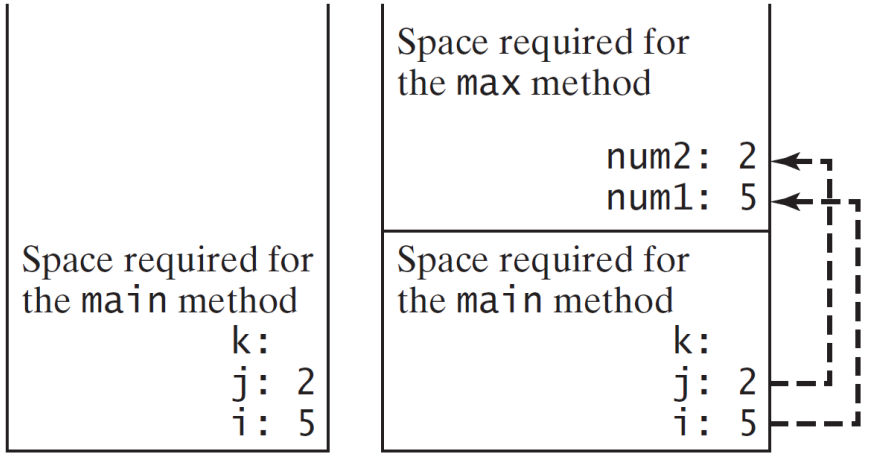
```
1  ...
2      // Method w/o return.
3      public static void display(int[] A) {
4
5          for (int i = 0; i < A.length; i++)
6              System.out.printf("%d ", A[i]);
7          System.out.println();
8
9      }
10
11     // Method returning array (reference)!
12     public static int[] arrayGen(int size, int low, int high) {
13
14         int[] A = new int[size];
15         int numOfStates = high - low + 1;
16         int offset = low;
17         for (int i = 0; i < A.length; i++)
18             A[i] = (int) (Math.random() * numOfStates) + offset;
19         return A;
20
21     }
22 ...
```

Method Invocation



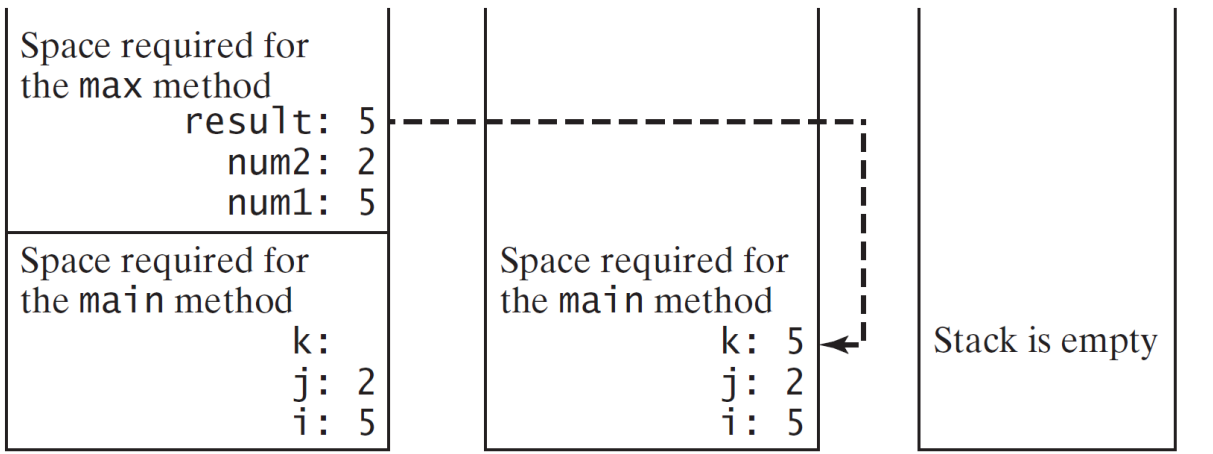
- Note that the input parameters are sort of variables declared within the method as placeholders.
- When calling the method, it's the obligation of callers to provide arguments in order, number, and compatible type, as defined in the method signature.

- In Java, method invocation uses pass-by-value.
- When the callee is invoked, the program control is transferred from the caller to the callee.
- For each method invocation, JVM pushes a frame which stores necessary information in the call stack.
- The caller resumes its work once the callee finishes its routine.



(a) The `main` method is invoked.

(b) The `max` method is invoked.



(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

Variable Scope

- A variable scope refers to the region where a variable can be referenced.
- A pair of balanced curly braces defines the variable scope.
- In general, variables can be declared in class level, method level, or loop level.
- If one local variable has its identifier identical to the class variable, then the local one is more preferable than the class one (i.e. ignore the latter).
  - This is called the shadow effect.

Example

```
1  public class ScopeDemo {
2
3      public static int x = 10; // Class level; global variable.
4
5      public static void main(String[] args) {
6
7          System.out.println(x); // Output 10.
8          int x = 100; // Method level, aka local variable.
9          x++;
10         System.out.println(x); // Output 101.
11         addOne();
12         System.out.println(x); // Output ?
13
14     }
15
16     public static void addOne() {
17
18         x = x + 1;
19         System.out.println(x); // Output ?
20
21     }
22 }
```

Math Toolbox: **Math** Class

- The class **Math** provides basic mathematical functions and two constants **Math**.PI and **Math**.E.
- All methods are public and static.
- You could refer to the official manual for **Math** here.
- As a professional engineer, you are expected to be able to read the manual![3]

[3] You may hear about RTFM: `https://en.wikipedia.org/wiki/RTFM`.

Special Issue: Method Overloading

- Name conflict is fine.
- Methods with the same name can coexist and be identified by method signatures.
- This can make programs clearer and more readable.

```
1 ...
2         public static int max(int x, int y) { ... }
3
4         // Different types.
5         public static double max(double x, double y) { ... }
6
7         // Different numbers of inputs.
8         public static int max(int x, int y, int z) { ... }
9 ...
```

Special Issue: Varargs

- JDK5 provides a short-hand for methods that support an arbitrary number of parameters of one type.

```
1  ...
2        /* You don't need to do these.
3        public static int max(int n1, int n2) { ... }
4        public static int max(int n1, int n2, int n3) { ... }
5        */
6
7        public static int max(int... nums) { ... }
8        // Equivalent to public static int max(int[] nums) { ... }
9
10       public static void main(String[] args) {
11
12         int x = max(100, 200, 300);
13         int y = max(100, 200, 300, 400);
14
15       }
16 ...
```

Special Issue: main(**String**[] args)

- I now could explain myself: the program starts to work by invoking the method main() together with a **String** array as the program parameters.

```
1 ...
2     public static void main(String[] args) {
3
4         for (String arg: args)
5             System.out.println(arg);
6
7     }
8 ...
```

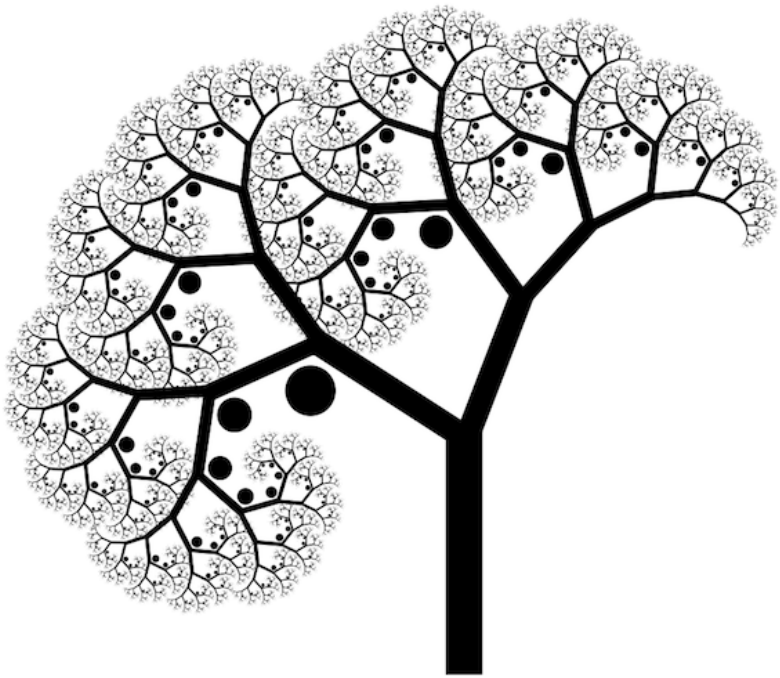
- In Eclipse, you may turn on the input dialog by adding "${string_prompt}" as a program argument to JVM.

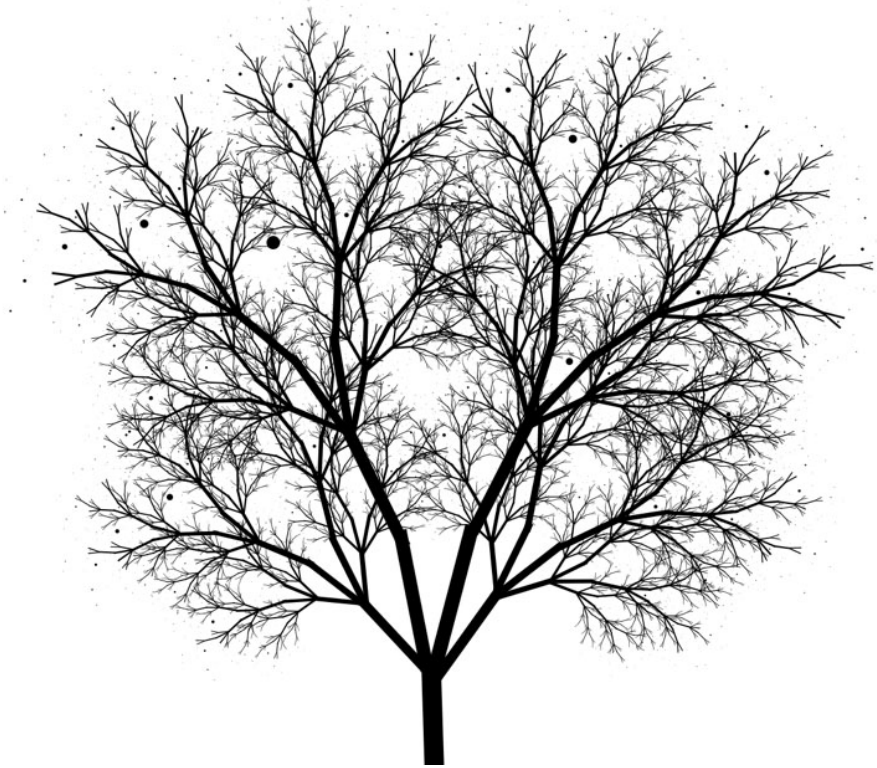
Recursion[4]

Recursion is a process of defining something in terms of itself.

- A method that calls itself is said to be recursive.
- Recursion is an alternative form of flow control.
- It is repetition without any loop.

[4] Recursion is a commom pattern in nature.





- Try Fractal.

Example: Factorial (Revisited)

- For example,

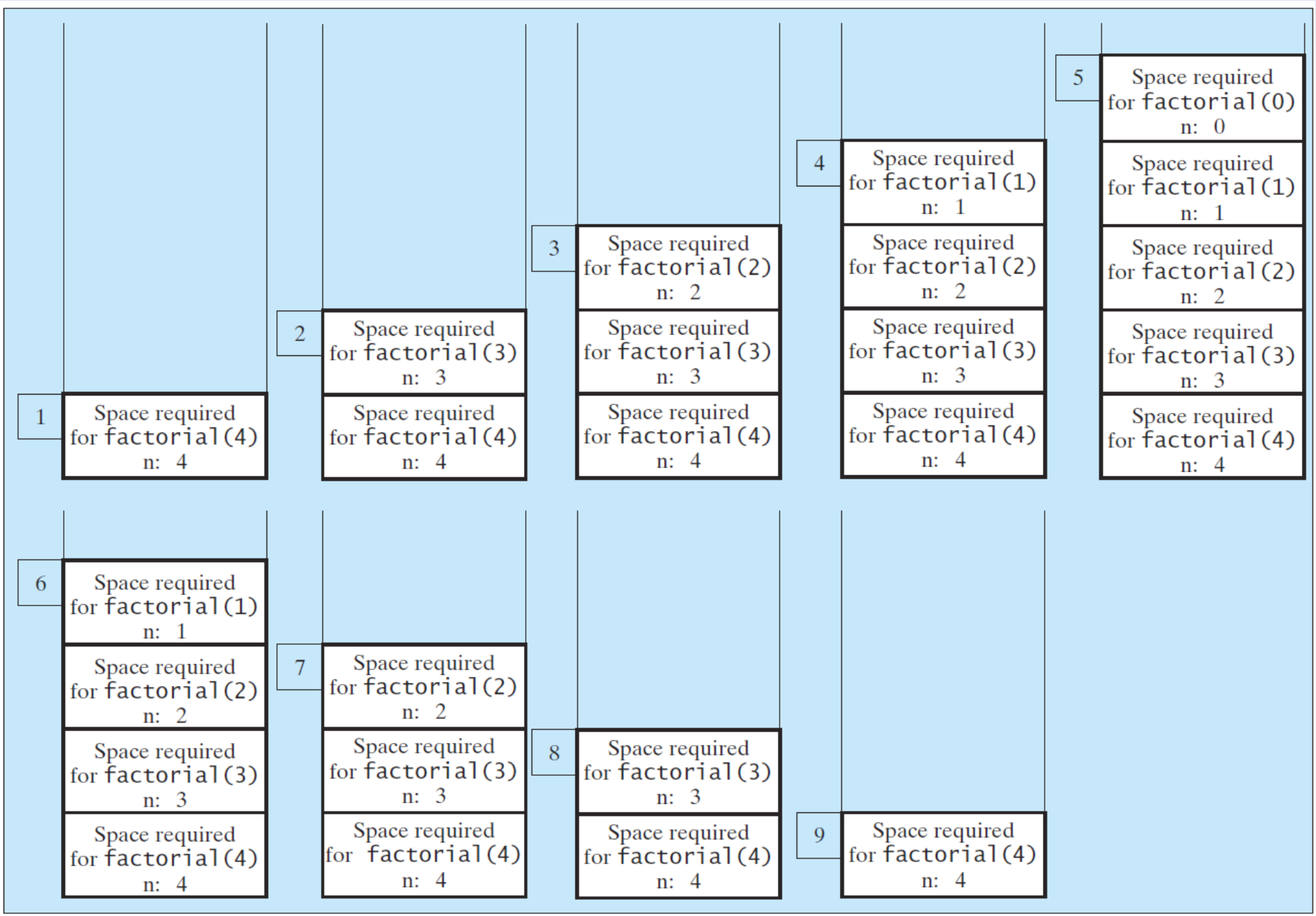
$$
\begin{aligned}
4! &= 4 \times 3 \times 2 \times 1 && \text{(in view of loops)} \\
&= 4 \times 3! && \text{(in view of recursion)} \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 24.
\end{aligned}
$$

- Find the pattern?

Write a program to determine $n!$ by recursion.

```
1 ...
2     public static int factorial(int n) {
3         if (n < 2)
4             return 1; // Base case.
5         else
6             return n * factorial(n - 1);
7     }
8 ...
```

- Remember to set a base case in recursion. (Why?)
- What is the time complexity?


1
Space required for factorial(4) n: 4
2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
3
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
4
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
5
Space required for factorial(0) n: 0
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
6
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
7
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
8
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
9
Space required for factorial(4) n: 4

```
1 ...
2         int s = 1;
3         for (int i = n; i > 1; i++) {
4             s *= i;
5         }
6 ...
```

- Both run in $O(n)$ time.
- One intriguing question is, Can we always turn a recursive method into a loop version of that?
- Affirmative.
- Church and Turing[5] proved that the loops and the recursions are equivalent.

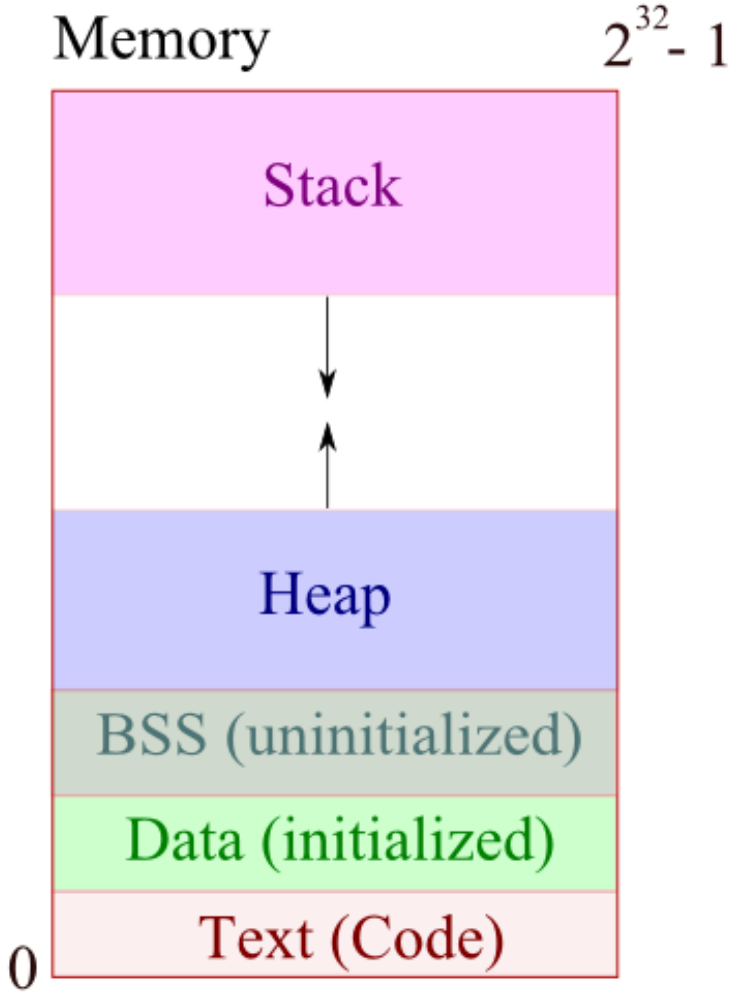
[5] See http://plato.stanford.edu/entries/church-turing/.

Remarks

- Recursion bears substantial overhead.
- So the recursive algorithm may execute a bit more slowly than the iterative equivalent.
- Moreover, a deep recursion depletes the call stack, which is limited, and causes a stack overflow[6] error.

[6] See `https://stackoverflow.com/`, `https://www.oreilly.com/`, and `https://www.quora.com/Does-reading-Copying-and-Pasting-from-Stack-Overflow-make-you-a-better`

Memory Layout



Exercise: Summation (Revisited)

Write a function to calculate the sum from 1 to n by recursion.

- For example, $n = 100$ and so we have

$$\begin{aligned} sum(100) &= 100 + sum(99) \\ &= 100 + 99 + sum(98) \\ &= 100 + 99 + 98 + sum(97) \\ &\vdots \\ &= 100 + 99 + 98 + \cdots + 1. \end{aligned}$$

- Can you find the recurrence relation?

```
...
    public static int sum(int n) {

        if (n == 1)
            return 1;
        else
            return n + sum(n - 1);

    }
...
```

```
...
    public static int sum(int n) {

        return n == 1 ? 1 : n + sum(n - 1);

    }
...
```

- Time complexity?

Exercise: Greatest Common Divisor (GCD)

Let a and b be two positive integers. Calculate GCD(a, b) by recursion.

- We proceed to implement the Euclidean algorithm.[7]
- For example,

$$\begin{aligned} \text{GCD}(54, 32) &= \text{GCD}(32, 22) \\ &= \text{GCD}(22, 10) \\ &= \text{GCD}(10, 2) \\ &= 2. \end{aligned}$$

[7] See `https://en.wikipedia.org/wiki/Euclidean_algorithm`.

```
...
    public static int gcd_by_recursion(int a, int b) {

        int r = a % b;
        if (r == 0)
            return b;
        return gcd_by_recursion(b, r); // Straightforward?!

    }
...
```

```
...
    public static int gcd_by_loop(int a, int b) {

        int r = a % b;
        while (r > 0) {
            a = b;
            b = r;
            r = a % b;
        }
        return b;

    }
...
```

Example: Fibonacci Numbers[8]

Let $n \geq 0$ be an integer. Calculate the n-th Fibonacci number F_n.

- Set $F_0 = 0$ and $F_1 = 1$.
- For $n > 1$, Fibonacci numbers can be found by

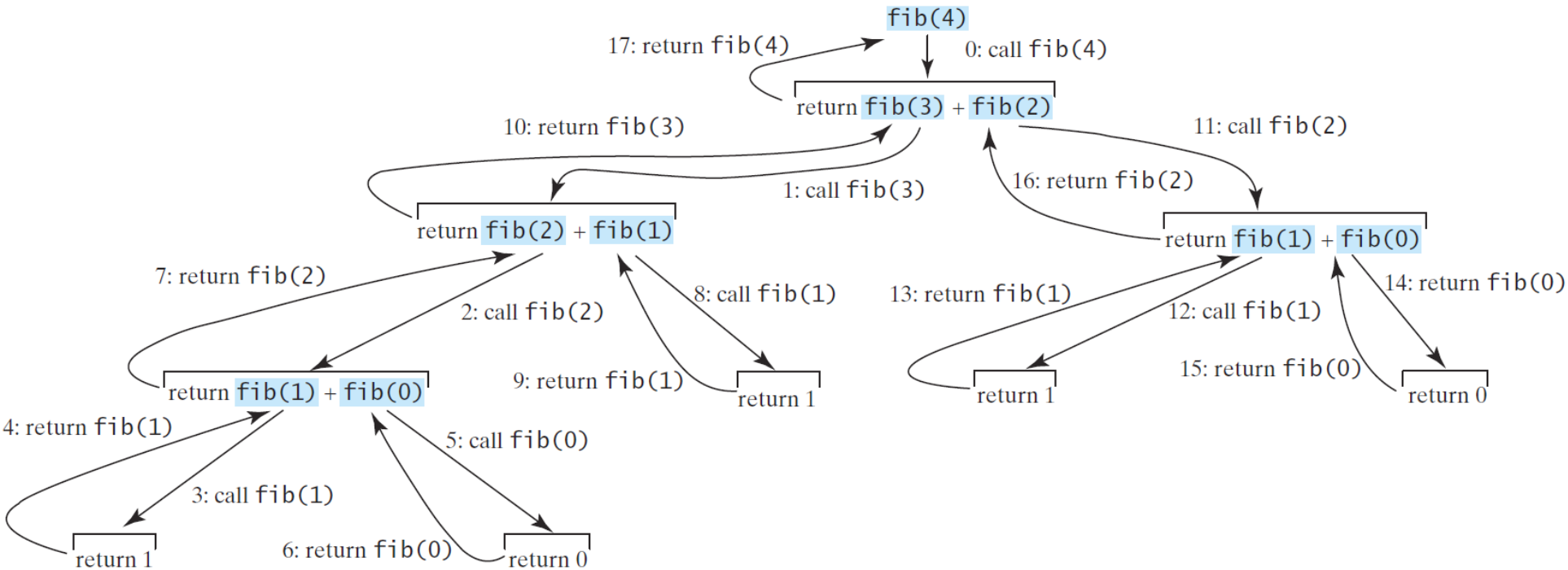
$$F_n = F_{n-1} + F_{n-2}.$$

- The first 10 numbers are as follows: $0, 1, 1, 2, 3, 5, 8, 13, 21,$ and 34.

[8] See `https://www.mathsisfun.com/numbers/fibonacci-sequence.html` and `https://en.wikipedia.org/wiki/Fibonacci_number`.

```
1  ...
2      public static int fib(int n) {
3
4          if (n < 2) {
5              return n;
6          } else {
7              return fib(n - 1) + fib(n - 2);
8          }
9
10     }
11 ...
```

- Short and clear!
- However, this algorithm suffers from poor performance!!
- Time complexity: $O(2^n)$. (Why!!!)


fib(4)
17: return fib(4)
0: call fib(4)
return fib(3) + fib(2)
10: return fib(3)
1: call fib(3)
11: call fib(2)
16: return fib(2)
return fib(2) + fib(1)
return fib(1) + fib(0)
7: return fib(2)
2: call fib(2)
8: call fib(1)
13: return fib(1)
12: call fib(1)
14: return fib(0)
return fib(1) + fib(0)
9: return fib(1)
return 1
return 1
15: return fib(0)
return 0
4: return fib(1)
5: call fib(0)
3: call fib(1)
return 1
6: return fib(0)
return 0

```
1  ...
2      public static double fib2(int n) {
3
4          if (n < 2) return n;
5
6          int x = 0, y = 1;
7          for (int i = 2; i <= n; i++) {
8              int z = x + y;
9              x = y;
10             y = z;
11         }
12         return y; // Why not z?
13
14     }
15 ...
```

- So it can be done in $O(n)$ time!
- The previous one (by recursion) is not optimal in time.
- Could you find a linear recursion for Fibonacci numbers?
- In fact, this problem can be done in $O(\log n)$ time!

Divide & Conquer

- We often use the divide-and-conquer strategy[9] to decompose the original problem into subproblems, which are more manageable.
  - For example, bubble sort.
- This benefits the program development as follows: easier to write, reuse, debug, modify, maintain, and also better to facilitate teamwork.

[9] Aka the stepwise refinement.

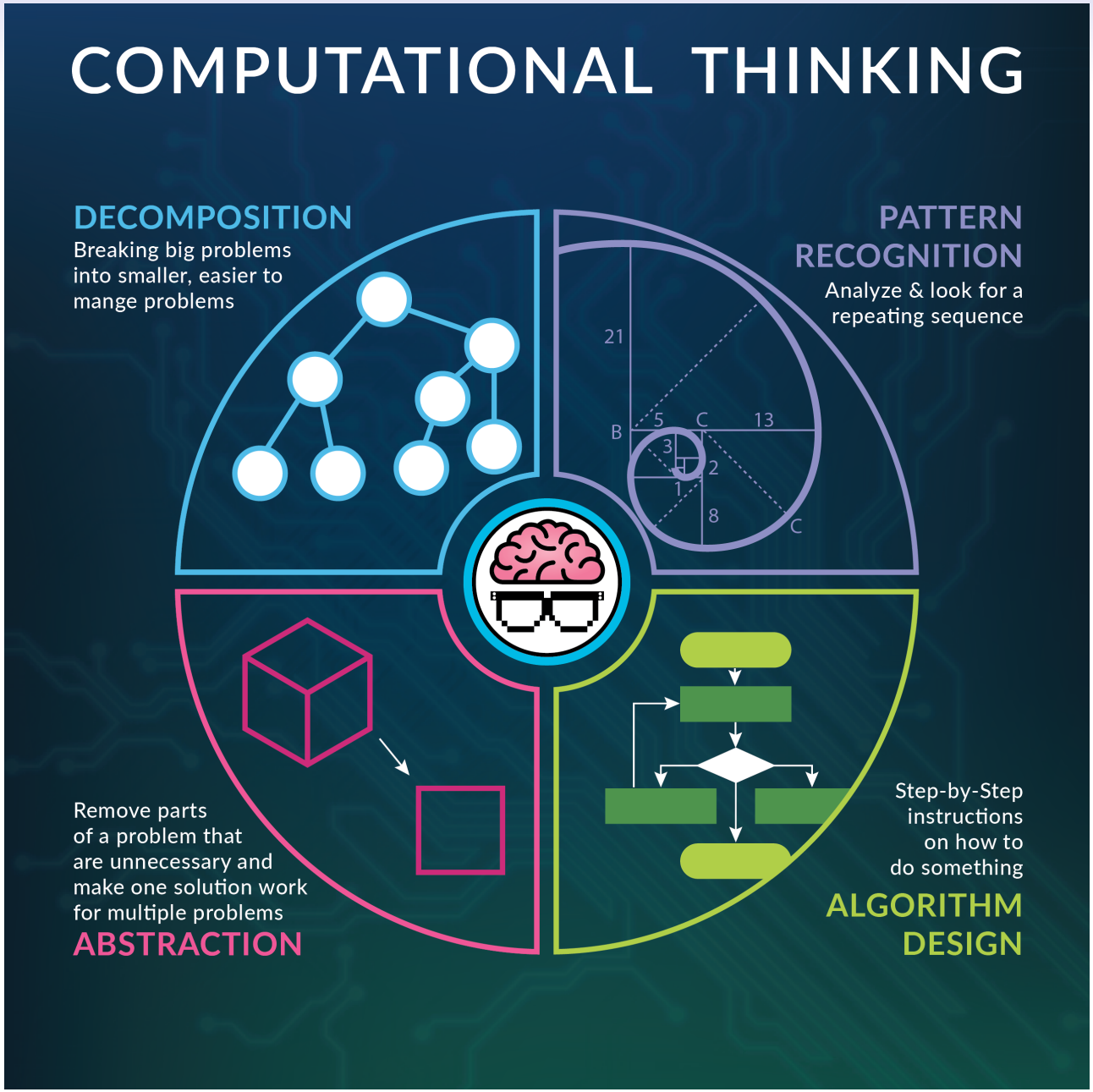
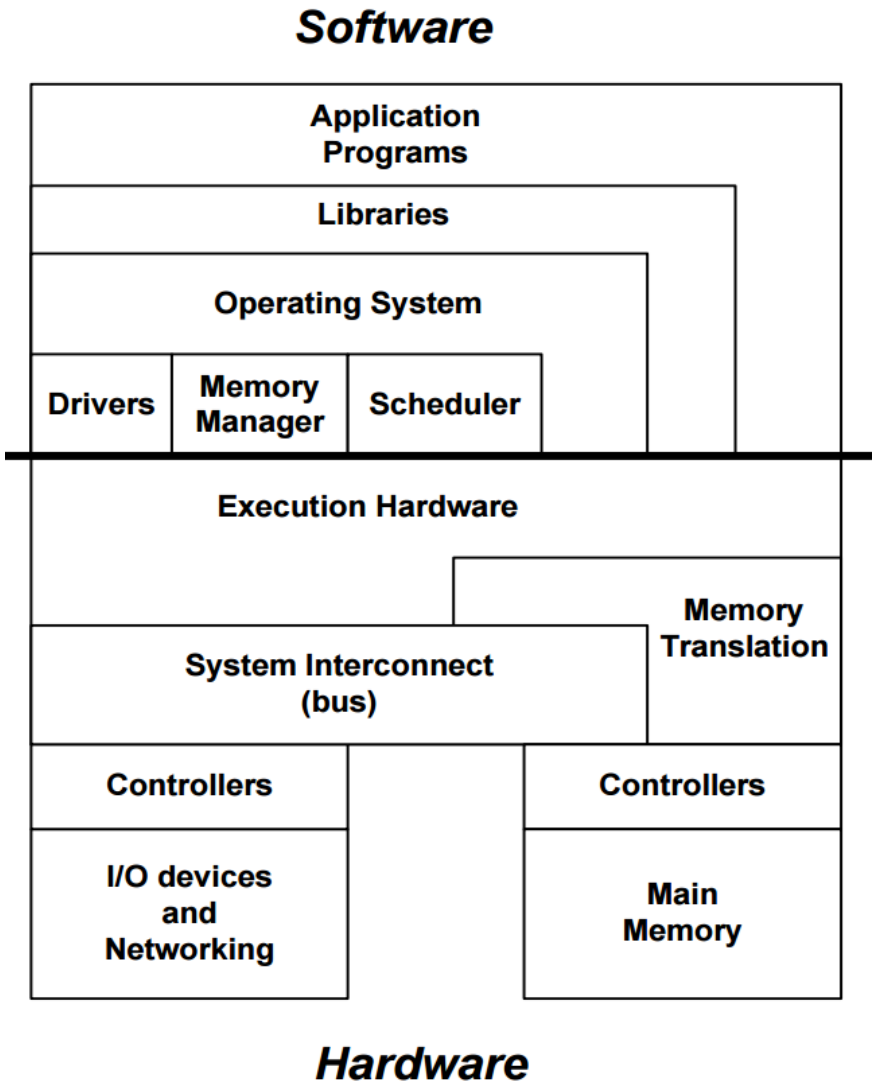


photo credit

Concept: Abstraction

- The abstraction process is to decide what details we need to highlight and what details we can ignore.
- Abstraction is everywhere.
  - An algorithm is an abstraction of a step-by-step procedure for taking input and producing output.
  - A programming language is an abstraction of a set of strings, each of which is interpreted to some computation.
  - And more.
- The abstraction process also introduces layers.
- Well-defined interfaces between layers enable us to build large and complex systems.

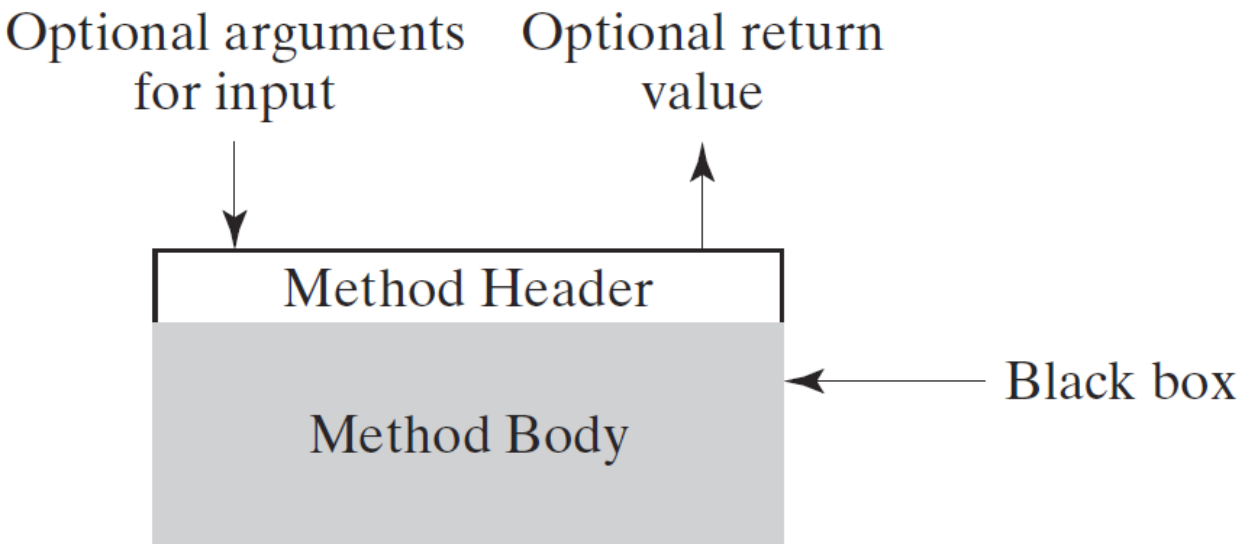
Example: Computer Systems



Example: Graphical User Interface (GUI)

- You have no idea about EM theory and communication systems; you know how to use the phone because you are familiar to the interface!

Example: Application Programming Interface (API)



- In building applications, an API simplifies programming by abstracting the underlying implementation and only exposing objects or actions the developer needs.

Concept: Abstraction (Concluded)

- As we have seen, methods/functions are control abstractions.
- Moreover, data structures like **ArrayList** are data abstractions.
- One can view the notion of an object as a way to combine abstractions of data and actions.
- Objects are everywhere.
- For example, describe about your cellphone.
  - Attributes: battery status, 4G signal, phonebook, album, music library, clips, and so on.
  - Functions? You can name it.