C# Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Online Course

```
class Lecture6
{
            "Methods and Recursion"
}

// Keywords:
return, ref, in, out, var, params
```

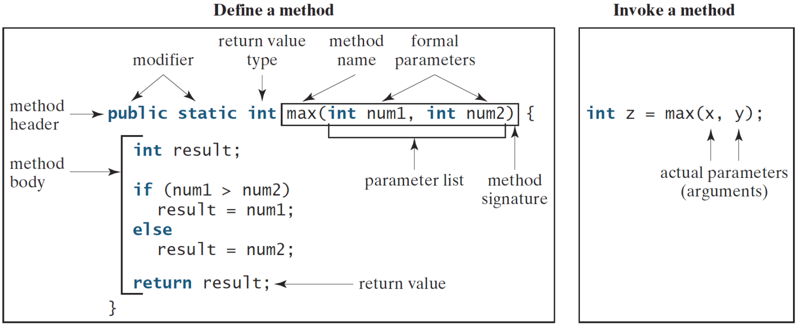
Methods

- Methods (or functions) can be used to define reusable code, so that it could organize and simplify the program.
- The concept of methods is similar to math functions, like

$$f(x, y),$$

  where x and y denote two input parameters.
- However, each input parameter should be declared with a specific type.
- Moreover, the method should be assigned with a return type before the method name!

Example: Max



Define a method

```
public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
}
```

Invoke a method

```
int z = max(x, y);
```

- The method signature comprises the method name and its parameter list.[1]

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

Alternatives?

```
...
    static int Max(int num1, int num2)
    {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
...
```

```
...
    static int Max(int num1, int num2)
    {
        return num1 > num2 ? num1 : num2;
    }
...
```

"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

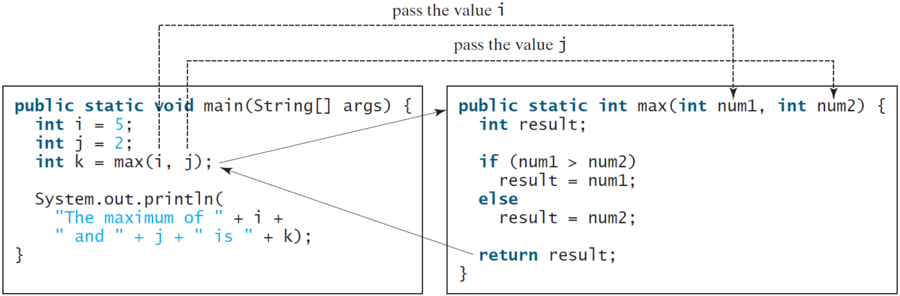
- The following two methods are incorrect.

```
...
    static int Foo1()
    {
        while (true);
        return 0; // Unreachable code. Nonsense?
    }

    static int Foo2(int x)
    {
        if (x > 0)
            return x; // What if x <= 0? Not allowed.
    }
...
```

More Examples

```
...
    // Method w/o return.
    static void Display(int[] A)
    {
        foreach (int item in A)
            Console.Write("{0} ", item);
        Console.WriteLine();
    }

    // Method returning array (reference)!
    static int[] ArrayFactory(int size, int low, int high)
    {
        int[] A = new int[size];
        Random rng = new Random();
        for (int i = 0; i < A.Length; i++)
            A[i] = rng.Next(low, high + 1);
        return A;
    }
...
```

Method Invocation

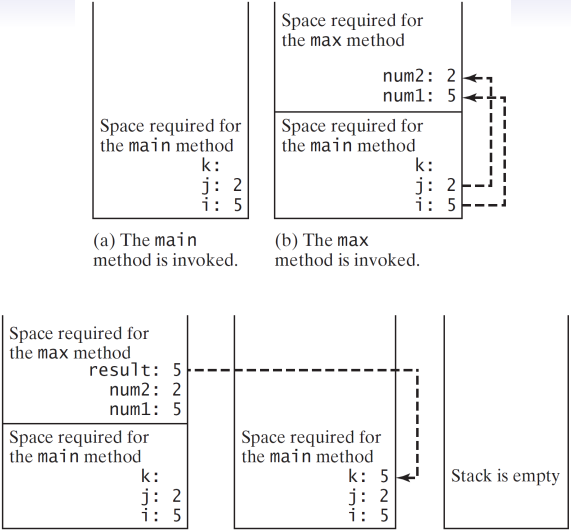


```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum of " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

- 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.

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



Space required for the main method
k:
j: 2
i: 5

(a) The main method is invoked.

Space required for the max method
num2: 2
num1: 5

Space required for the main method
k:
j: 2
i: 5

(b) The max method is invoked.

Space required for the max method
result: 5
num2: 2
num1: 5

Space required for the main method
k:
j: 2
i: 5

(c) The max method is being executed.

Space required for the main method
k: 5
j: 2
i: 5

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

Stack is empty

(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

```
...
    static int x = 10; // Class level; global variable.

    static void Main(string[] args)
    {
        int x = 100; // Method level, aka local variable.
        x = x + 1;
        Console.WriteLine(x); // Output 101.
        AddOne();
        Console.WriteLine(x); // Output ?
    }

    static void AddOne()
    {
        x = x + 1;
        Console.WriteLine(x); // Output ?
    }
...
```

Alternative: Pass-By-Reference[2]

- The ref keyword indicates a value that is passed by reference.
- This makes the formal parameter an alias for the argument.
- Any operation on this formal parameter is directly applied to the referencee!

[2] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref`.

```
...
    static void Main(string[] args)
    {
        int x = 100;
        x = x + 1;
        Console.WriteLine(x); // Output 101.
        AddOne(ref x);
        Console.WriteLine(x); // Output 102.
    }

    static void AddOne(ref int x)
    {
        x =  x + 1;
        Console.WriteLine(x); // Output 102.
    }
...
```

About ref & out

- Using the in keyword does not allow the callee to modify the argument value, as known as read-only protection!
- The out keyword is similar to ref and in, except that both require variable initialization before they are passed.
- However, the called method using out is required to assign a value before the method returns.

Exercise

```
...
    static void Main(string[] args)
    {
        string text = "528";

        int number;
        if (Int32.TryParse(text, out number))
            Console.WriteLine("Converted {0} to {1}.", text, number);
        else
            Console.WriteLine("Unable to convert {0}.", text);
    }
...
```

- The variable *number* is used as the output variable from the method.
- In this sense, this syntax offers one possibility for multiple returns!

Special Issue: Implicitly Typed Local Variables[4]

- Local variables can be declared without an explicit type.[3]
- The var keyword means that the compiler infers and assigns the most appropriate type, for example,

```
...
        var i = 5;                    // i is compiled as an int.
        var A = new[] { 0, 1, 2 };    // A is compiled as int[].
        foreach (var item in A)
        {
            Console.WriteLine("{0}", item);
        }
...
```

[3] You could use var only in methods and loops.

[4] See https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables.

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.
- Note that methods cannot have signatures that differ only by ref, in, or out.

```
...
    static int Max(int x, int y) { ... }

    // Different types.
    static double Max(double x, double y) { ... }

    // Different numbers of inputs.
    static int Max(int x, int y, int z) { ... }
...
```

Special Issue: params

```
...
    // You don't need to do these below.
    // static int Max(int n1, int n2) { ... }
    // static int Max(int n1, int n2, int n3) { ... }

    static int Max(params int[] nums) { ... }

    static void Main(string[] args)
    {
        int x = max(100, 200, 300);
        int y = max(100, 200, 300, 400);
    }
...
```

Special Issue: Optional Arguments

- Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.
- Each optional parameter has a default value as part of its definition.
- If no argument is sent for that parameter, the default value is used.

```
...
    static void DoAction(int a,
                         double b = 10.0,
                         string c = "default") { ... }
...
```

Special Issue: Main() & Command-Line Arguments[6]

- Recall that the Main method is the entry point of C# applications.
- Note that there can only be one entry point in C# programs.
- You may use PowerShell to start your C# program with some parameters.[5]

```
...
    static void Main(string[] args)
    {
        foreach (var arg in args)
            Console.WriteLine(arg);
    }
...
```

[5] See https://docs.microsoft.com/en-us/powershell/.

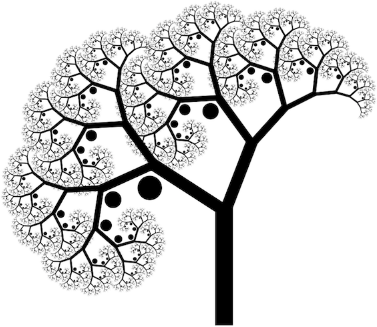
[6] See https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/main-and-command-args/.

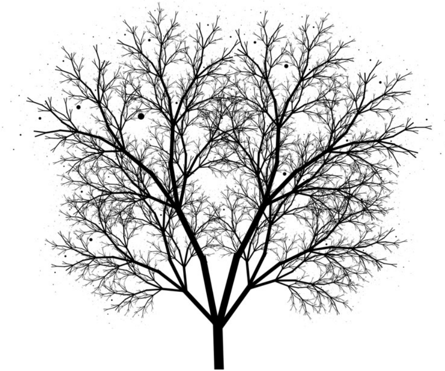
Recursion[7]

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.

[7] Recursion is a commom pattern in nature.





- Try Fractal.

Example: Factorial (Revisited)

- For example,

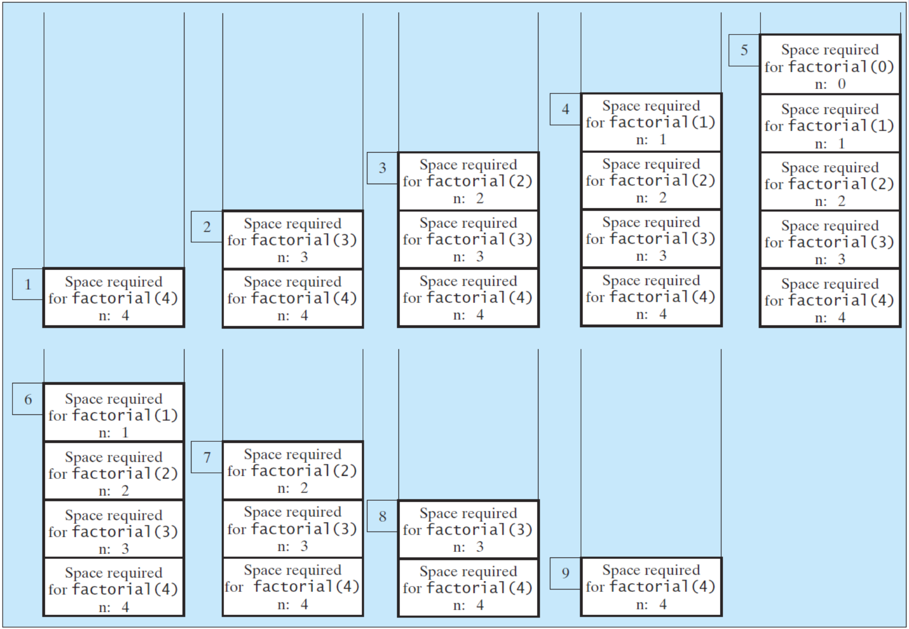
$$\begin{aligned} 4! &= 4 \times 3 \times 2 \times 1 \text{ (in view of loops)} \\ &= 4 \times 3! \qquad\qquad \text{(in view of recursion)} \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times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

- Find the pattern?

Write a program to determine n! by recursion.

```
...
    static int Factorial(int n)
    {
        if (n < 2)
            return 1;                       // Base case.
        else
            return n * Factorial(n - 1); // Recurrence relation.
    }
...
```

- 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

```
...
        int s = 1;
        for (int i = n; i > 1; i--)
        {
            s *= i;
        }
...
```

- 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[8] proved that the loops and the recursions are equivalent.

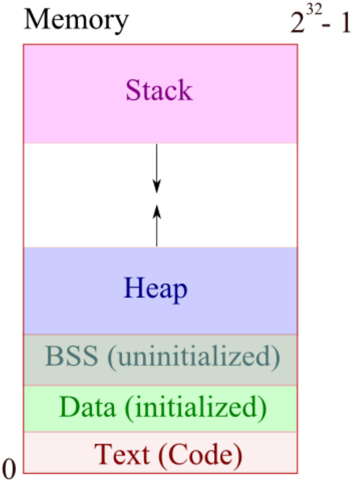
[8] See http://plato.stanford.edu/entries/church-turing/.

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[9] error.

[9] 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?

```
...
    static int Sum(int n)
    {
        if (n == 0)
            return 0;
        else
            return n + Sum(n - 1);
    }
...
```

```
...
    static int Sum(int n)
    {
        return n == 0 ? 0 : 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.[10]
- For example,

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

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

```
...
    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?!
    }
...
```

```
...
    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[11]

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

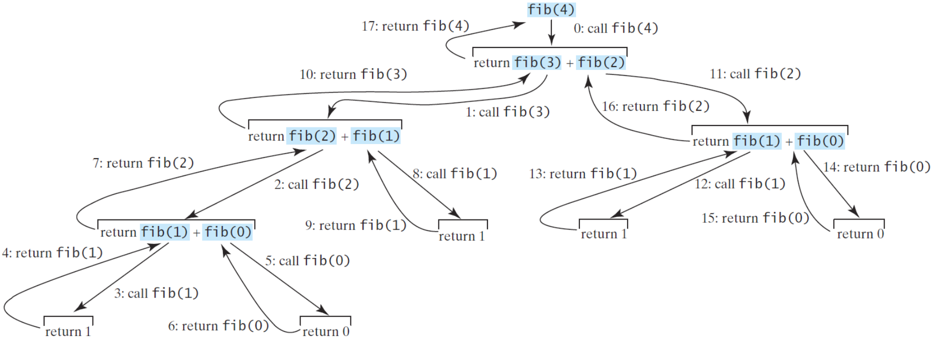
$$F_n = F_{n-1} + F_{n-2}.$$

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

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

```
...
    static int Fib(int n)
    {
        if (n < 2)
            return n;
        else
            return Fib(n - 1) + Fib(n - 2);
    }
...
```

- 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)

11: call fib(2)

1: call fib(3)

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)

9: return fib(1)

15: return fib(0)

return fib(1) + fib(0)

return 1

return 1

return 0

4: return fib(1)

5: call fib(0)

3: call fib(1)

6: return fib(0)

return 1

return 0

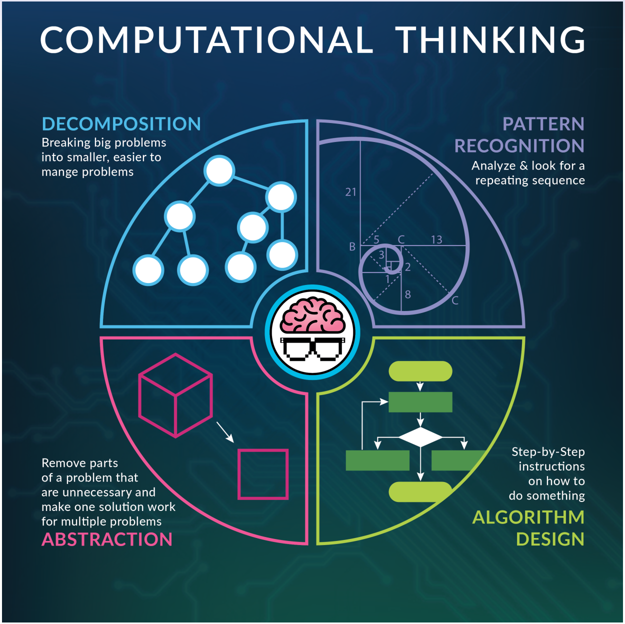
```
...
    static double Fib2(int n)
    {
        if (n < 2) return n;

        int x = 0, y = 1;
        for (int i = 2; i <= n; i++)
        {
            int z = x + y;
            x = y;
            y = z;
        }
        return y; // Why not z?
    }
...
```

- 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[12] 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.

[12] Aka the stepwise refinement.



COMPUTATIONAL THINKING

DECOMPOSITION

Breaking big problems into smaller, easier to mange problems

PATTERN RECOGNITION

Analyze & look for a repeating sequence

Remove parts of a problem that are unnecessary and make one solution work for multiple problems

ABSTRACTION

Step-by-Step instructions on how to do something

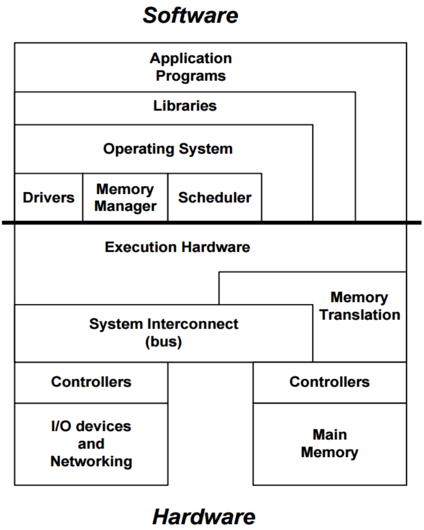
ALGORITHM DESIGN

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.

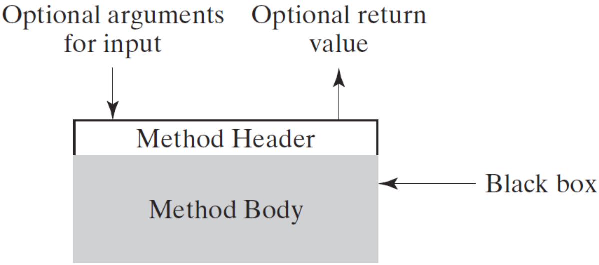
Example: Computer Systems



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 **Array** (denoted by [ ]) 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.