Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java 405
Spring 2024

```java
class Lecture5 {

                "Methods and Recursion"

}

// Keywords:
return, var
```

Introduction

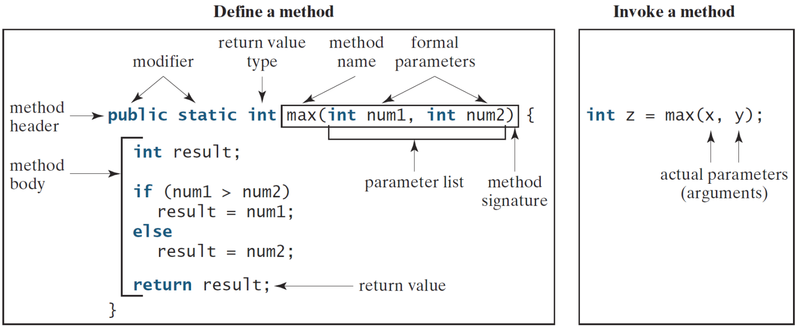
- Methods[1] are used to define reusable codes, so that it could organize and simplify your programs.
- The idea of methods originates from math, like

$$f(x, y),$$

  where x and y are its two input parameters.
- Every parameter should be declared with one specific type.
- Every method needs one return type even if it has no return!

[1] As known as functions, procedures and subroutine.

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 its name and parameter list.

Alternatives?

```java
...
    public static int max(int num1, int num2) {

        if (num1 > num2) {
            return num1;
        } else {
            return num2;
        }

    }
...
```

```java
...
    public 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 the return Statement

- The return statement terminates the method.
- A caller invokes one method, called the callee.
- The caller and the callee should follow the method header, like a contract.
- The caller provides the callee with adequate inputs and receives one return value from the callee (or none if the return type is void).
- Note that a method could have more than one return statement.

Pitfalls

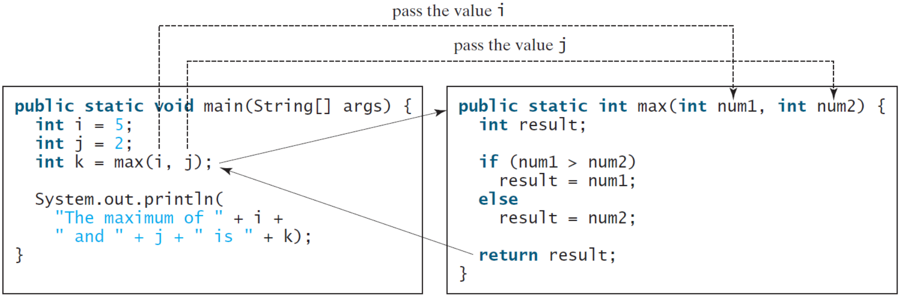
```
...
    public static int foo1() {

        while (true);
        return 0; // Unreachable code.

    }

    public static int foo2(int x) {

        if (x > 0)
            return x; // What if x <= 0? Not allowed.

    }
...
```

- Stick to the contract!

More Examples

```java
...
    // Method w/o return.
    public static void display(int[] A) {

        for (int item : A)
            System.out.printf("%d ", item);
        System.out.println();

    }

    // Method returning array (reference)!
    public static int[] arrayFactory(int size, int low, int high) {

        int[] A = new int[size];
        int numOfStates = high - low + 1;
        for (int i = 0; i < A.length; i++)
            A[i] = (int) (Math.random() * numOfStates) + low;
        return A;

    }
...
```

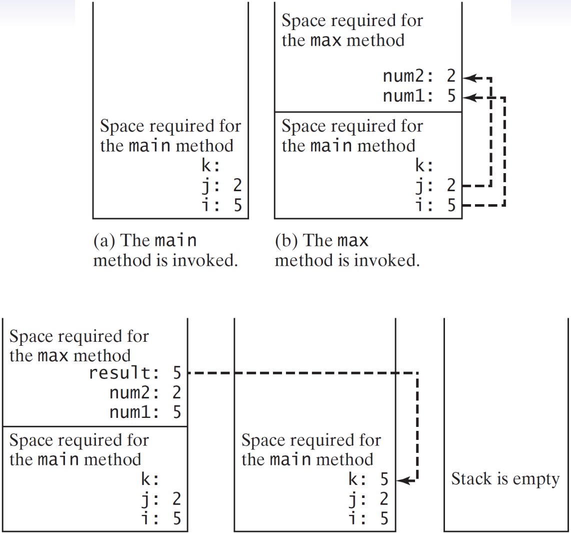
Method Invocation



```
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;
}
```

- The formal parameters are sort of variables declared within the method as placeholders.
- When invoking the method, the caller passes (copies) the arguments to the callee, in order and compatible type.
- This is called passing by value.


Space required for the main method
k:
j: 2
i: 5
Space required for the max method
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
Space required for the max method
result: 5
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
Space required for the main method
k: 5
j: 2
i: 5
Stack is empty

(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.

- The JVM pushes a frame into the call stack[2], which stores the arguments and other necessary information for each method invocation.
- Once the method reaches any return statement (or the bottom of the method), the frame will be nullified and the JVM returns to where the caller jumps from.
- It also implies that the memory space occupied by the frame will be recycled for next method invocation.
- Note that the execution flow of method invocation is the central concept of recursions.

[2] A data structure with the first-in-last-out (FILO) property is called a stack.

Variable Scope

- A variable scope is the region where one variable is visible.
- A variable has one of the following three scopes: class level[3], (method) local level, and loop level[4].
- As a local variable, any changes made inside the method does not affect the original value.
- Note that one local variable can have its name identical to the one of class level.
- This is called the shadow effect because we favor the local one. (Why?)

[3] We will discuss about this kind later in the next chapter.

[4] We've discussed the loop variables in the chapter of flow control: Any variable declared in the loop is invisible when the loop is finished.

Example

```java
public class ScopeDemo {

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

    public static void main(String[] args) {

        System.out.println(x); // Output 10.
        int x = 100; // Method level, aka local variable.
        x++;
        System.out.println(x); // Output 101.
        addOne();
        System.out.println(x); // Output? Why?

    }

    public static void addOne() {

        x = x + 1;
        System.out.println(x); // Output?

    }
}
```

Local Variable Type Inference[6]

```java
...
        var x = 100;                       // x will be an integer.
        var y = "type inference";  // y will be a string.
        var z = new ArrayList<>(); // z will be an ArrayList object.
...
```

- Type inference is a compiler's ability to automatically infer unspecified data type parameters from contextual information.
- It allows us to write more concise Java code when it comes to generics and lambda expression[5].
- Note that this is applicable only for local variables.

[5] We will meet lambda expressions soon.

[6] Added in JDK 10. See Java 10 Local Variable Type Inference.

Manual for Math Toolbox: **Math** Class

- The **Math** class provides basic math functions and two global constants **Math**.PI and **Math**.E.
- Check out the official document for **Math**.[7]
- As you can see, its methods are all public and static.
- As a professional programmer, you should be capable to read documents (manuals) to survive in the future![8]

[7] See Math from Oracle's official document.

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

Method Overloading

- Naming conflict is allowed when methods with the same name can be identified by method signatures.

```
...
        public static int max(int x, int y) { ... }

        // Differ in types.
        public static double max(double x, double y) { ... }

        // Differ in numbers of inputs.
        public static int max(int x, int y, int z) { ... }
...
```

- Note that this mechanism does not relate to the return type.

Special Issue: Variadic Functions[9]

```
...
    // You don’t have to do these below:
    // public static int max(int n1, int n2) { /* Ignored */ }
    // public static int max(int n1, int n2, int n3) { /* Ignored */

    public static int max(int... nums) { /* Ignored */ }
    // The above method definition is equivalent to
    // public static int max(int[] nums) { /* Ignored */ }

    public static void main(String[] args) {

        int x = max(100, 200, 300);
        int y = max(100, 200, 300, 400);

    }
...
```

- The ellipsis (...) allows the user to pass an arbitrary number of arguments to the method.

[9] Since JDK5. It is one of syntactic sugars.

The Entry Method: main(**String**[] args)

- You can start the program together with a series of strings.
- Those attached strings are stored in one **String** array as the program parameters.

```
...
    public static void main(String... args) {

        for (String arg : args)
            System.out.println(arg);

    }
...
```

- In Eclipse, you may turn on the input dialog by adding “${string_prompt}“ as a program argument to JVM.
- You can also compile and run the program in the command line interface (CLI).

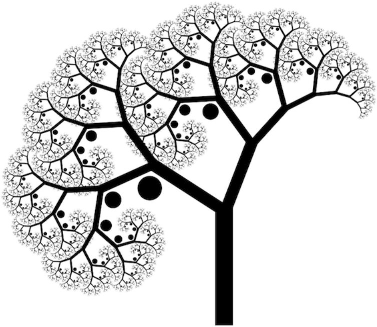
Recursion[10]

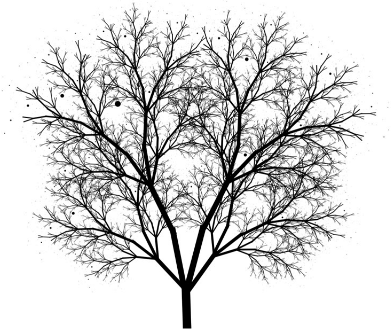
Recursion is a process of defining something in terms of itself.

- A method that calls itself in some way is recursive.
- Recursion is an alternative form of repetition without any loop.

[10] Recursion is a commom pattern in nature.

Examples of Natural World





- Try Fractal.

Example: Factorial (Revisited)

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

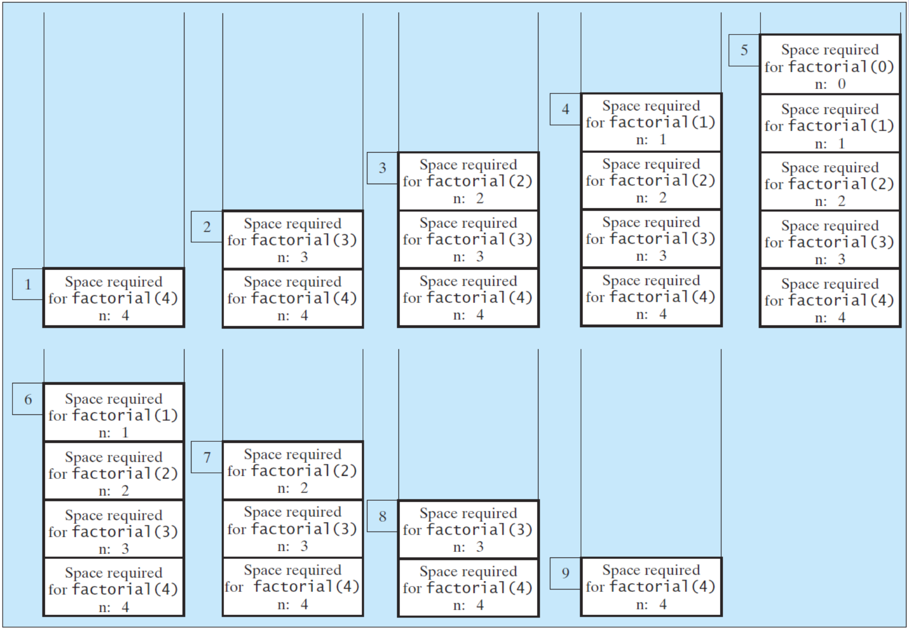
- For example,

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

- Find any recursive pattern?

```java
...
    public 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

```java
...
        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.
    - The Church-Turing Thesis[11] implies that both are equivalent.

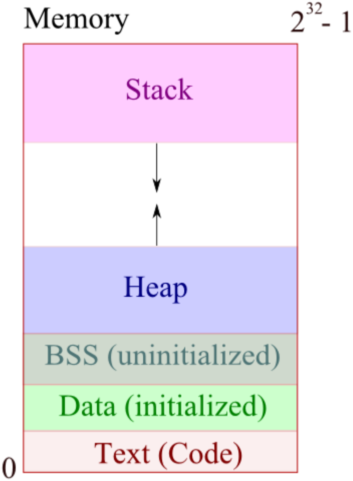
[11] See http://plato.stanford.edu/entries/church-turing/.

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 the error **StackOverflowError**.[12]

[12] See https://stackoverflow.com/, https://www.oreilly.com/, and https://www.quora.com/Does-reading-Copying-and-Pasting-from-Stack-Overflow-mak

Memory Layout



Exercise: Summation (Revisited)

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

- For example,

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

- Can you find the recurrence relation?

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

        if (n == 1)
            return 1;
        return 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 implement the Euclidean algorithm for GCD.[13]
- For example,

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

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

```java
...
    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?!

    }
...
```

```java
...
    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 Sequence[14]

Let n be a nonnegative integer. Calculate the n-th Fibonacci number F_n.

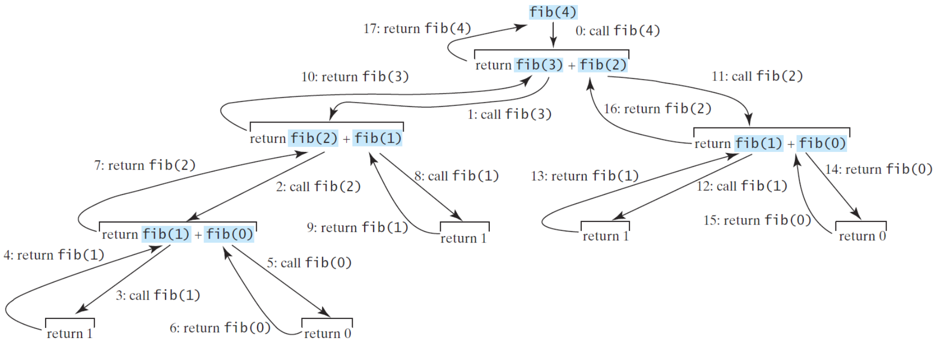
- Set $F_0 = 0$ and $F_1 = 1$.
- For $n > 1$, the Fibonacci numbers follows the recurrence relation
$$F_n = F_{n-1} + F_{n-2}.$$
- The first 10 numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21,$ and 34.

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

```java
...
    public static int fib(int n) {

        if (n < 2) {
            return n;
        } else {
            return fib(n - 1) + fib(n - 2);
        }

    }
...
```

- Time complexity: $O(2^n)$. (Why!!!)
- This algorithm suffers from the performance issue.
  - Assume that the modern CPU can finish 10^9 times of method invocation per second.
  - Then it takes 36.6 years for F_{60}.[15]

[15] You could reproduce the number by calculating $2^{60}/(10^9 \times 86400)$.



- A binary tree with height level h has at most $2^h - 1$ nodes. (Why?)
- The algorithm runs in $O(2^n)$ time because its execution counts grow like a binary tree.
- Can we do better by avoiding recomputaions?

```
...
    public 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?

    }
...
```

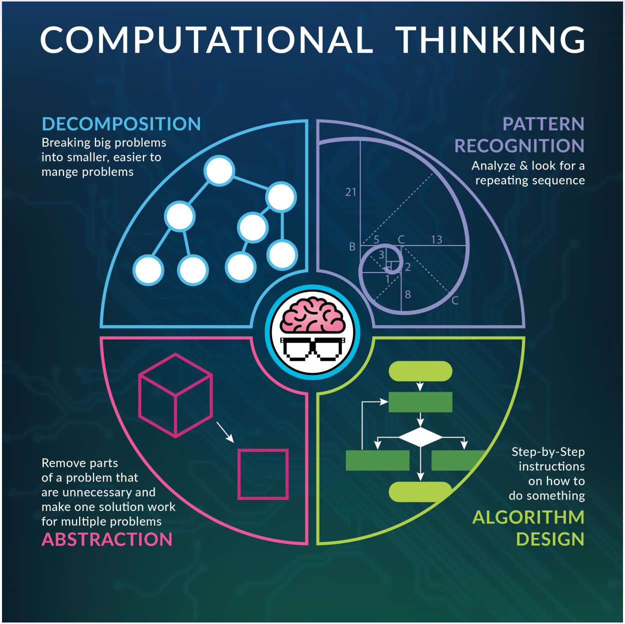
- The algorithm runs in $O(n)$ time!
- Could you find the $O(n)$-time recursive one?
- In fact, this problem can be solved in $O(\log n)$ time![16]

[16] See 509. Fibonacci Number of LeetCode.

Problem-Solving Skill: Divide & Conquer (DC)

- We often use the DC strategy to decompose the original problem into several manageable subproblems.
- It is also similar to do a study: narrow down to one doable topic and solve it.
- This approach benefits the program development, say easier to write, more possible to reuse, and better to facilitate teamwork.
- One more thing to note is that one method should not exceed, in principle, 20 lines of codes.[17]

[17] See Clean Code.



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

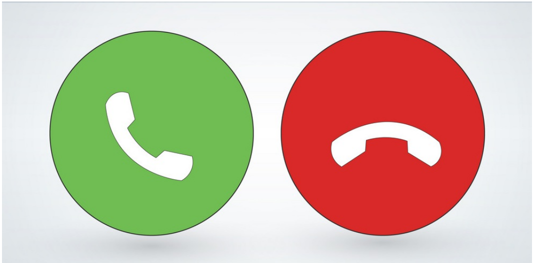
Step-by-Step instructions on how to do something

ALGORITHM DESIGN

Programming Concept: Abstraction

- Abstraction provides an interface to application programmers that separates policy from mechanism.
  - Policy: what the interface commits to accomplishing.
  - Mechanism: how the interface is implemented.
- This process enables us to build large and complex systems.
- Abstraction is everywhere, even in everyone's daily life.
- You can find a lot of similar experiences about abstraction.
  - For example, driving a car, writing Java programs.

Example: Graphical User Interface (GUI)



- You probably have no idea about electromagnetism and communication systems.
- However, you know how to make a phone call because you are familiar to its user interface!

Conclusions

- Methods are control abstractions while data structures are data abstractions.
- We can treat the notion of objects as a way to combine data and control abstractions.
- For example, try to enumerate the data with its associated controls in your cellphone.
  - Data: phone book, photo album, music library, clips, etc.
  - Controls? The buttons you can press in those apps.
- We will start with the object-oriented programming (OOP) paradigm in the next chapter.

"Abstraction is selective ignorance."

"We can solve any problem by introducing an extra level of indirection."

– Andrew Koenig (1952–)

"Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

– Edsger Dijkstra (1930–2002)