# Jump Statements

The keyword break and continue are often used in repetition structures to provide additional controls.

- break: the loop is terminated right after a break statement is executed.
- continue: the loop skips this iteration right after a continue statement is executed.
- In practice, jump statements in loops should be conditioned.

# Example: Primality

Write a program which determines if the input integer is a prime number.

- Let $x > 1$ be any natural number.
- Then $x$ is said to be a prime number if $x$ has no positive divisors other than 1 and itself.
- It is then straightforward to check if it is prime by dividing $x$ by all natural numbers smaller than $x$.
- For speedup, you can divide $x$ by only numbers smaller than $\sqrt{x}$. (Why?)

```
1  ...
2          Scanner input = new Scanner(System.in);
3          System.out.println("Enter x > 2?");
4          int x = input.nextInt();
5          boolean isPrime = true;
6          input.close();
7
8          double upperBd = Math.sqrt(x);
9          for (int y = 2; y < upperBd; y++) {
10             if (x % y == 0) {
11                 isPrime = false;
12                 break;
13             }
14         }
15
16         if (isPrime) {
17             System.out.println("Prime");
18         } else {
19             System.out.println("Composite");
20         }
21 ...
```

# Exercise (Revisited)

- Redo the cashier problem by using an infinite loop with a break statement.

```
1 ...
2         while (true) {
3             System.out.println("Enter price?");
4             price = input.nextInt();
5             if (price <= 0) break;
6             total += price;
7         }
8         System.out.println("Total = " + total);
9 ...
```

# Another Example: Compounding

Write a program which determines the holding years for an investment doubling its value.

- Let *balance* be the current amount, *goal* be the goal of this investment, and $r$ be the annual interest rate.
- Then this investment should take at least $n$ years so that the balance of the investment can double its value.
- Recall that the compounding formula is given by

$$balance = balance \times (1 + r/100).$$

```
1  ...
2          int r = 18; // 18%
3          int balance = 100;
4          int goal = 200;
5
6          int years = 0;
7          while (balance <= goal) {
8              balance *= (1 + r / 100.0);
9              years++;
10         }
11
12         System.out.println("Balance = " + balance);
13         System.out.println("Years = " + years);
14 ...
```

```
...
        int years = 0; // should be declared here; scope issue
        for (; balance <= goal; years++) {
            balance *= (1 + r / 100.0);
        }
...
```

```
...
        int years = 1; // check this initial value
        for (; true; years++) {
            balance *= (1 + r / 100.0);
            if (balance > goal) break;
        }
...
```

- A for loop can be an infinite loop by setting true or simply leaving empty in the condition statement.
- An infinite for loop with an if-break statement is equivalent to a normal while loop.

# Equivalence: while and for Loops (Concluded)

In general, a for loop may be used if the number of repetitions is known in advance. If not, a while loop is preferred.

# Nested Loops

A loop can be nested inside another loop.

- Nested loops consist of an outer loop and one or more inner loops.
- Each time the outer loop is repeated, the inner loops are reentered, and started anew.

# Example

## Multiplication table

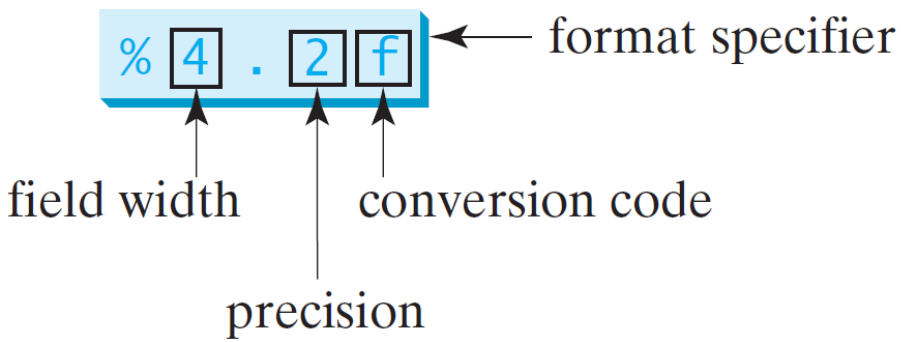
Write a program which displays the multiplication table.

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
3  6  9 12 15 18 21 24 27
4  8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

# Formatting Console Output

You can use *System.out.printf*() to display formatted output on the console.

```
1 ...
2         double amount = 1234.601;
3         double interestRate = 0.00528;
4         double interest = amount * interestRate;
5         System.out.printf("Interest = %4.2f", interest);
6 ...
```

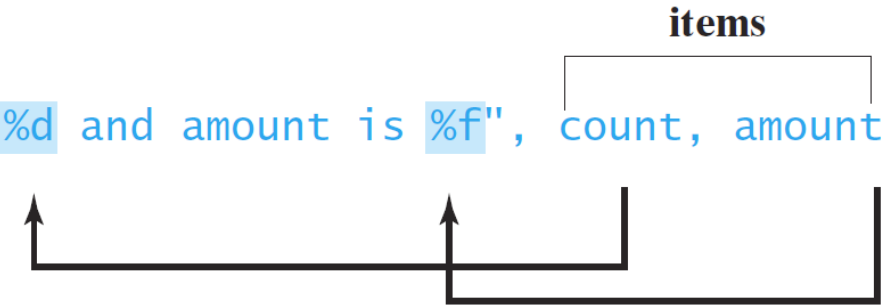


| *Format Specifier* | *Output* | *Example* |
|---|---|---|
| %b | a Boolean value | true or false |
| %c | a character | 'a' |
| %d | a decimal integer | 200 |
| %f | a floating-point number | 45.460000 |
| %e | a number in standard scientific notation | 4.556000e+01 |
| %s | a string | "Java is cool" |

- By default, a floating-point value is displayed with 6 digits after the decimal point.

# Multiple Items to Print

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

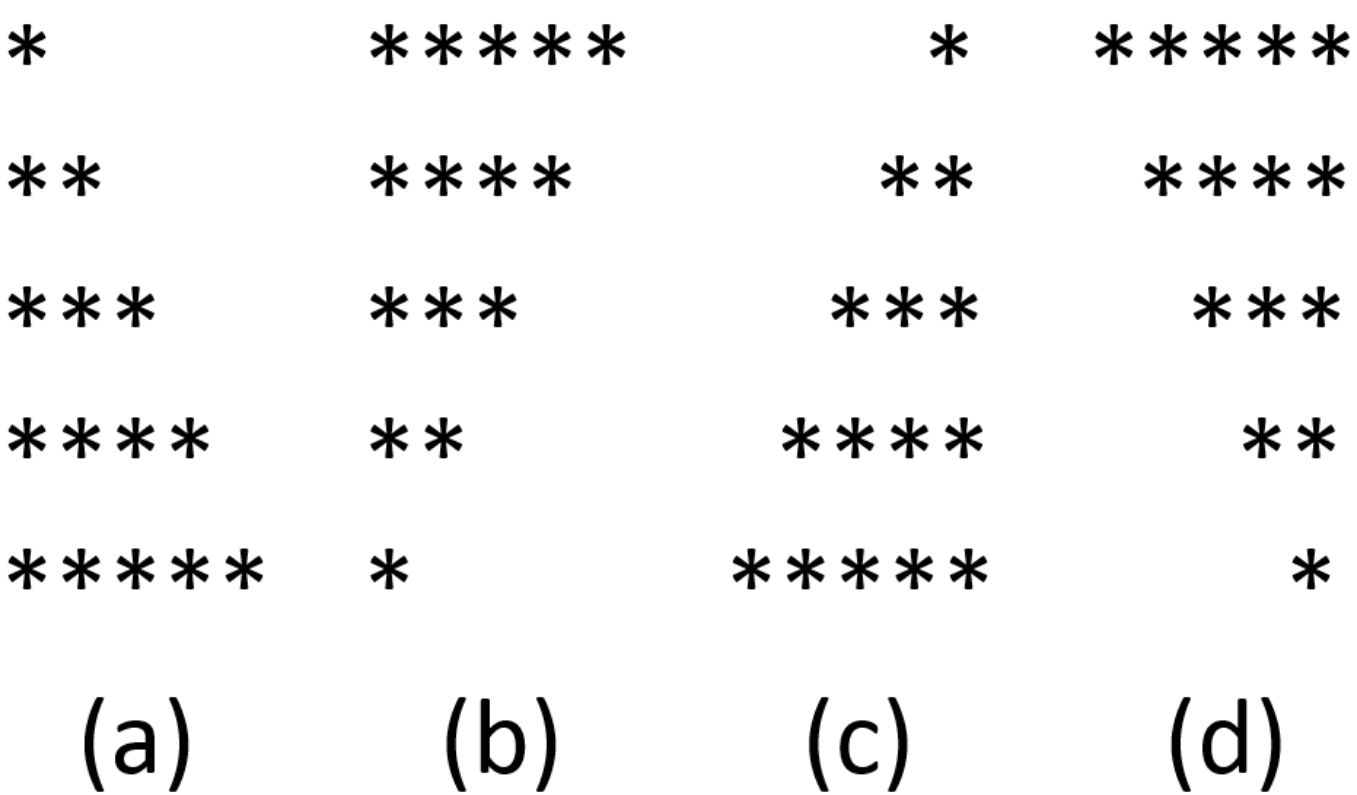



```
display                count is 5 and amount is 45.560000
```

- Items must match the format specifiers in order, in number, and in exact type.
- If an item requires more spaces than the specified width, the width is automatically increased.
- By default, the output is right justified.
- You may try the plus sign (+), the minus sign (-), and 0 in the middle of format specifiers.
  - Say %+8.2f, %-8.2f, and %08.2f.

```
...
    public static void main(String[] args) {
        for (int i = 1; i <= 9; ++i) {
            for (int j = 1; j <= 9; ++j) {
                System.out.printf("%3d", i * j);
            }
            System.out.println();
        }
    }
...
```

# Exercise: Coupled Loops



```java
public class PrintStarsDemo {
    public static void main(String[] args) {
        // case (a)
        for (int i = 1; i <= 5; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.printf("*");
            }
            System.out.println();
        }

        // case (b), (c), (d)
        // your work here
    }
}
```

# Analysis of Algorithms

- First, there may exist some algorithms for the same problem.
- Then we compare these algorithms.
- The first question is, Which one is more efficient? (Why?)
- We focus on the growth rate of the running time or space requirement as a function of the input size $n$, denoted by $f(n)$.

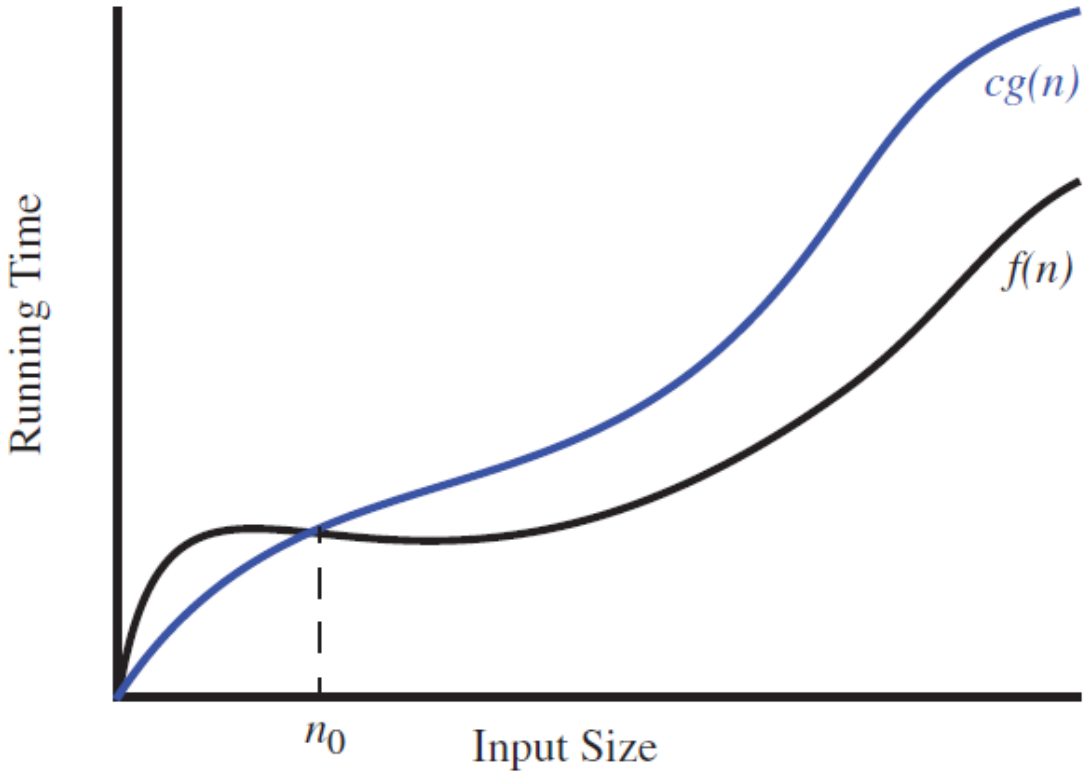
# $O$-notation[1]

- In math, $O$-notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.
- $f(n) \in O(g(n))$ as $n \to \infty$ if and only if there is a constant $c > 0$ and a real number $n_0$ such that

$$|f(n)| \leq c|g(n)| \quad \forall n \geq n_0. \tag{1}$$

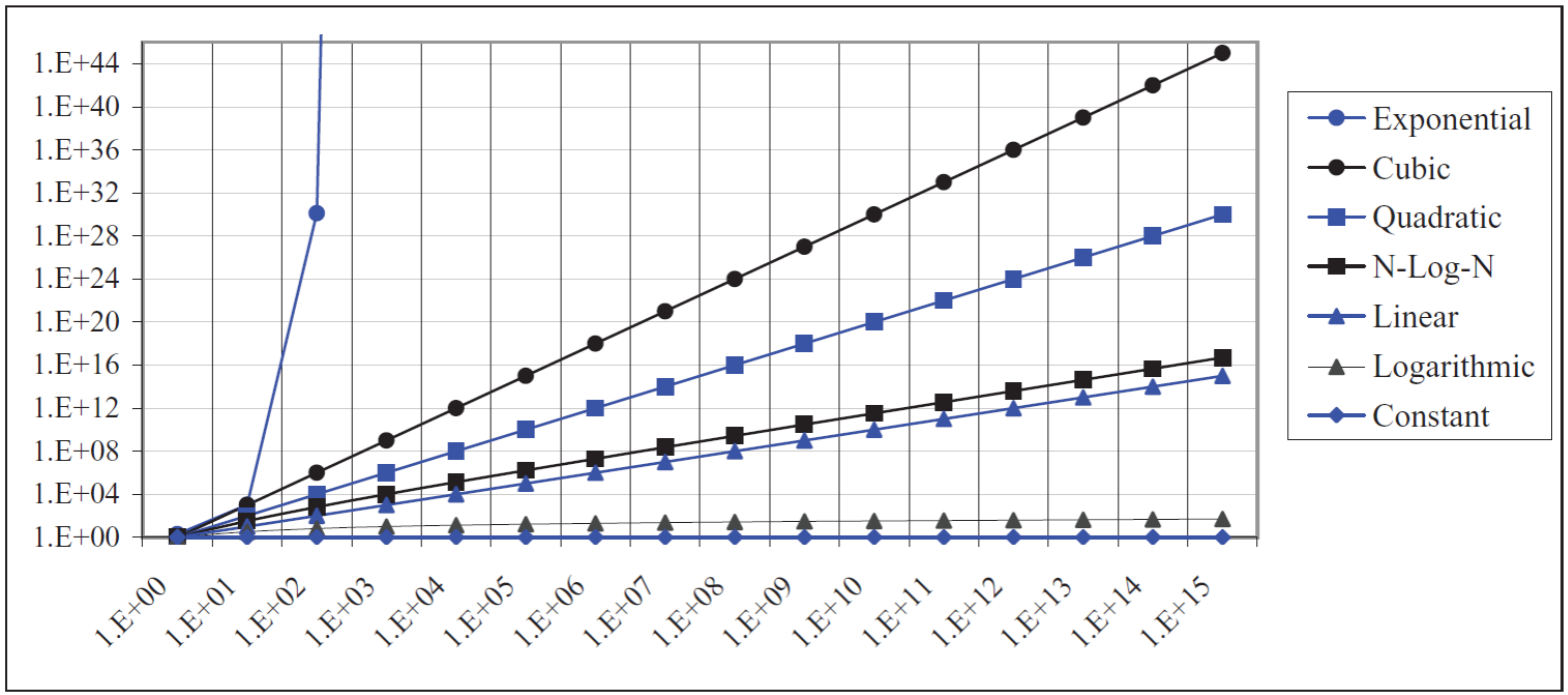
- Note that $O(g(n))$ is a set featured by some simple function $g(n)$.
- Hence $f(n) \in O(g(n))$ is equivalent to say that $f(n)$ is one instance of $O(g(n))$.

[1] See any textbook for data structures and algorithms or `https://en.wikipedia.org/wiki/Big_O_notation`.




- For example, $8n^2 - 3n + 4 \in O(n^2)$.
- We could say that $8n^2 - 3n + 4 \in O(n^3)$ and $8n^2 - 3n + 4 \notin O(n)$.

# Common Fundamental Functions[2]




| *constant* | *logarithm* | *linear* | *n-log-n* | *quadratic* | *cubic* | *exponential* |
|---|---|---|---|---|---|---|
| 1 | $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $a^n$ |

[2]See Table 4.1 and Figure 4.2 in Goodrich and etc, p. 161.

- We use $O$-notation to describe the asymptotic[3] upper bound of complexity of the algorithm.
- So $O$-notation is widely used to classify algorithms by how they respond to changes in its input size.[4]
  - Time complexity
  - Space complexity
- Note that we often make a trade-off between time and space.
  - Unlike time, we can reuse memory.

[3] The asymptotic sense is that the input size $n$ grows toward infinity.

[4] Actually, there are $\Theta, \theta, o, \Omega$, and $\omega$ which are used to classify algorithms.

# References

- https://en.wikipedia.org/wiki/Game_complexity
- https://en.wikipedia.org/wiki/P_versus_NP_problem

```
1 class Lecture5 {
2
3                     "Arrays"
4
5 }
```

# Arrays

An array stores a large collection of data which is of the same type.

```
1 ...
2         // assume the size variable exists above
3         T[] A = new T[size];
4         // this creates an array of T type, referenced by A
5 ...
```

- **T** can be any data type.
- This statement comprises two parts:
  - Declaring a reference
  - Creating an array

# Variable Declaration for Arrays

- In the left-hand side, it is a declaration for an array variable, which does not allocate real space for the array.
- In reality, this variable occupies only a certain space for the reference to an array.[5]
- If a reference variable does not refer to an array, the value of the variable is null.[6]
- In this case, you cannot assign elements to this array variable unless the array object has already been created.

[5] Recall the stack and the heap in the memory layout.

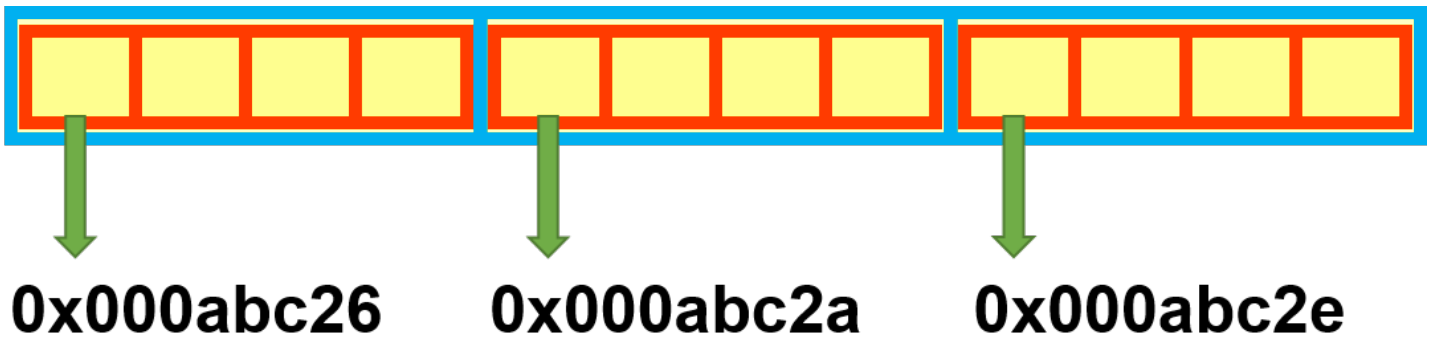
[6] Moreover, this holds for any reference variable. For example, the **Scanner** type.

# Creating A Real Array

- All arrays of Java are objects.
- As seen before, the new operator returns the memory address of that object.
  - Recall that the type of reference variables must be compatible to that of the array object.
- The variable *size* must be a positive integer for the number of elements.
- Note that the size of an array cannot be changed after the array is created.[7]

[7] Alternatively, you may try the class **ArrayList**, which is more useful in practice.

# Array in Memory




```
1        int[] A = new int[3];
```

- The array is allocated contiguously in the memory.
- All arrays are zero-based indexing.[8] (Why?)
- So we have A[0], A[1], and A[2].

[8] Same in C, C++, python, Javascript, and more.

# Array Initializer

The elements of arrays are initialized once created.

- By default, every element is assigned as follows:
  - 0 for all numeric primitive data types
  - \u0000 for char type
  - false for boolean type
- An array can also be initialized by enumerating all the elements without using the new operator.
- For example,

```
1        int[] A = {1, 2, 3};
```

# Processing Arrays

When processing array elements, we often use for loops.

- Recall that arrays are objects.
- They have an attribute called **length** which records the size of the arrays.
  - For example, use A.length to get the size of A.
- Since the size of the array is known, it is natural to use a for loop to manipulate with the array.

# Many Examples

## Initialization of arrays by a Scanner object

```
1 ...
2         // let x be an integer array with a certain size
3         for (int i = 0; i < A.length; ++i) {
4             A[i] = input.nextInt();
5         }
6 ...
```

## Initialization of arrays by random numbers

```
1 ...
2         for (int i = 0; i < A.length; ++i) {
3             A[i] = (int) (Math.random() * 10);
4         }
5 ...
```

## Display of array elements

```
1 ...
2         for (int i = 0; i < A.length; ++i) {
3             System.out.printf("%3d", A[i]);
4         }
5 ...
```

## Sum of array elements

```
1 ...
2         int sum = 0;
3         for (int i = 0; i < A.length; ++i) {
4             sum += A[i];
5         }
6 ...
```

## Extreme values in the array

```
1 ...
2         int max = A[0];
3         int min = A[0];
4         for (int i = 1; i < A.length; ++i) {
5             if (max < A[i]) max = A[i];
6             if (min > A[i]) min = A[i];
7         }
8 ...
```

- How about the location of the extreme values?
- Can you find the 2nd max of A?
- Can you keep the first $m$ max of A?

## Shuffling over array elements

```
1  ...
2          for (int i = 0; i < A.length; ++i) {
3              // choose j randomly
4              int j = (int) (Math.random() * A.length);
5              // swap
6              int tmp = A[i];
7              A[i] = A[j];
8              A[j] = tmp;
9          }
10 ...
```

- How to swap values of two variables without *tmp*?
- However, this naive algorithm is biased.[9]

[9] See https://blog.codinghorror.com/the-danger-of-naivete/.

# Exercise

## Deck of Cards

Write a program which picks first 5 cards at random from a deck of 52 cards.

- 4 suits: Spade, Heart, Diamond, Club
- 13 ranks: $3, \ldots, 10$, J, Q, K, A, 2
- Label 52 cards by $0, 1, \cdots, 51$
- Shuffle the numbers
- Deal the first 5 cards

```
...
        String[] suits = {"Spade", "Heart", "Diamond", "Club"};
        String[] ranks = {"3", "4", "5", "6" , "7",
                          "8", "9", "10", "J", "Q", "K",
                          "A", "2"};

        int size = 52;
        int[] deck = new int[size];
        for (int i = 0; i < deck.length; i++)
            deck[i] = i;

        // shuffle over deck; correct version
        for (int i = 0; i < size - 1; i++) {
            int j = (int) (Math.random() * (size - i)) + i;
            int z = deck[i];
            deck[i] = deck[j];
            deck[j] = z;
        }

        for (int i = 0; i < 5; i++) {
            String suit = suits[deck[i] / 13];
            String rank = ranks[deck[i] % 13];
            System.out.printf("%8s%3s\n", suit, rank);
        }
...
```

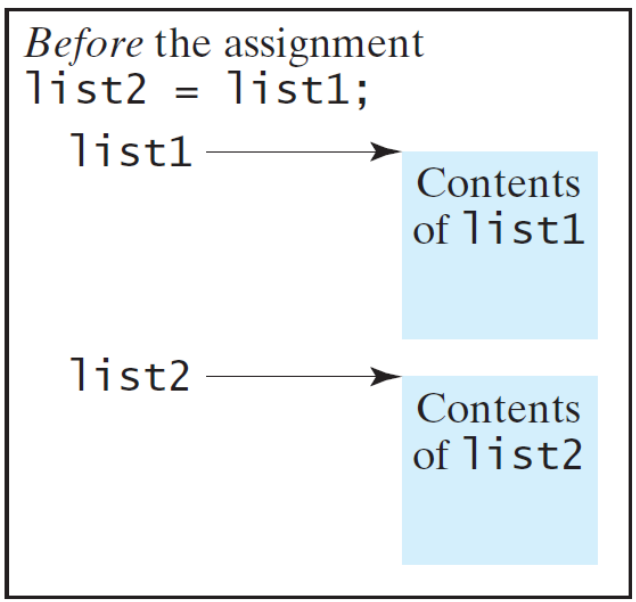
# Cloning Arrays

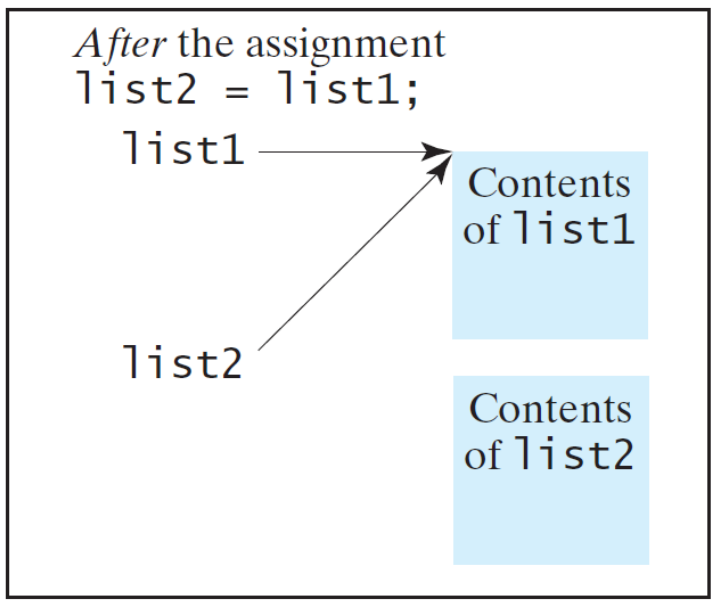
- In practice, one might duplicate an array for some reason.
- One could attempt to use the assignment statement (=), for example,

```
1 ...
2         T[] A = {...}; // assume A is an array
3         T[] B = A; // shallow copy; you don't have a new array
4 ...
```

- However, this is impossible to make two distinct arrays.
- Recall that the array variables are simply references to the arrays in the heap.

- Moreover, all the reference variables share this property!
- For example,



- Use a loop to copy individual elements one by one.

```
...
        int[] A = {2, 1, 3, 5, 10};
        int[] B = new int[A.length];
        // deep copy
        for (int i = 0; i < A.length; ++i) {
            B[i] = A[i];
        }
...
```

- Alternatively, you may use the *arraycopy* method in the **System** class.

```
...
        int[] A = {2, 1, 3, 5, 10};
        int[] B = new int[A.length];
        System.arraycopy(A, 0, B, 0, A.length);
...
```

# for-each Loops[10]

- A for-each loop is designed to iterate over a collection of objects, such as arrays and other data structures, in strictly sequential fashion, from start to finish.
- For example,

```
1 ...
2         T[] A = {...}; // assume some T-type array
3         for (T element: A) {
4             // body
5         }
6 ...
```

- Note that the type **T** should be compatible to the element type of *A*.

[10] Beginning with JDK5. Now we have JDK9.

# Example

```
1 ...
2         int[] A = {1, 2, 3};
3         int sum = 0;
4         for (int i = 0; i < A.length; ++i)
5             sum += A[i];
6 ...
```

- Not only is the syntax streamlined, but it also prevents boundary errors.

```
1 ...
2         int[] A = {1, 2, 3};
3         int sum = 0;
4         for (int x: A)
5             sum += x;
6 ...
```

# Short Introduction to Data Structures

- A data structure is a particular way of organizing data in a program so that it can be used efficiently.
- Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations.
- In comparison, a data structure is a concrete implementation of the specification provided by some ADT.
- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.[11]

[11] See `http://bigocheatsheet.com/`.

# Common Operations on Data

- A specific data structure is chosen in one problem.
- Then the operations are implemented accordingly.
- The **Arrays** class contains useful methods for common array operations such as sorting and searching.
- For example,

```
1 import java.util.Arrays;
2
3 ...
4         int[] A = {5, 2, 8};
5         Arrays.sort(A); // sort the whole array
6
7         char[] B = {'A', 'r', 't', 'h', 'u', 'r'};
8         Arrays.sort(B, 1, 3); // sort the array partially
9 ...
```

# Selection Sort

```
1  ...
2          // selection sort
3          for (int i = 0; i < A.length; i++) {
4              int k = i; // the position of min starting from i
5              for (int j = i + 1; j < A.length; j++) {
6                  if (A[k] > A[j])
7                      k = j;
8              }
9              // swap(A[i], A[k])
10             int tmp = A[k];
11             A[k] = A[i];
12             A[i] = tmp;
13         }
14 ...
```

- Time complexity: $O(n^2)$
- You can find more sorting algorithms.[12]

[12] See `http://visualgo.net/`.

# Linear Search

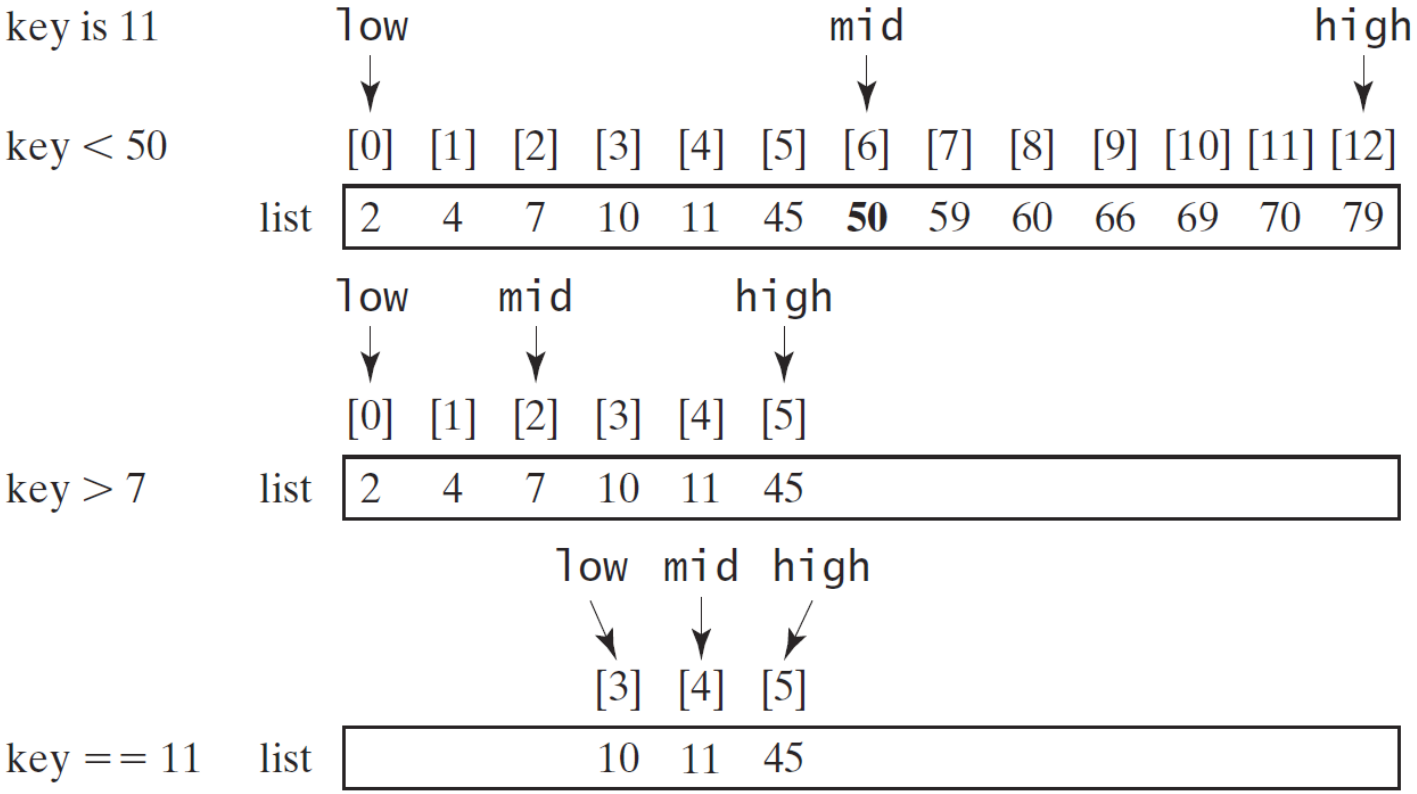
Write a program which searches for the index associated with the key.

- For convenience, assume that there is no duplicate key.
- The linear search approach compares the key with each element in the array sequentially.

```
1 ...
2         // assume A is an array
3         // linear search
4         for (int i = 0; i < A.length; i++) {
5             if (A[i] == key) {
6                 System.out.printf("%3d", i);
7             }
8         }
9 ...
```

- Time complexity: $O(n)$

# Alternative: Binary Search



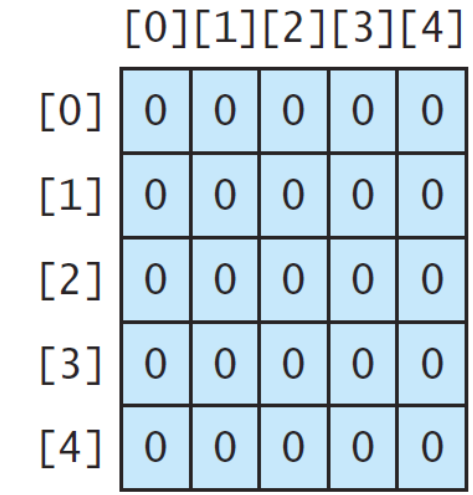

- Time complexity: $O(\log n)$
- Overall time complexity (sorting + searching): still $O(\log n)$?

```
1  ...
2          int index = -1; // why?
3          int high = A.length - 1, low = 0, mid;
4          while (high > low) {
5              mid = (high + low) / 2;
6              if (A[mid] == key) {
7                  index = mid;
8                  break;
9              } else if (A[mid] > key)
10                 high = mid - 1;
11             else
12                 low = mid + 1;
13         }
14
15         if (index > -1)
16             System.out.printf("%d: %d\n", key, index);
17         else
18             System.out.printf("%d: does not exist\n", key);
19 ...
```

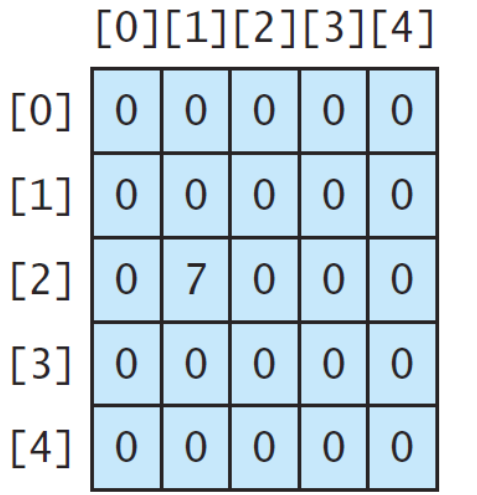
# Beyond 1-Dimensional Arrays

- 2D or high-dimensional arrays are widely used.
  - For example, a colorful image is represented by three 2D arrays (R, G, B).
- We can create a 2D **T**-type array with 4 rows and 3 columns as follows:

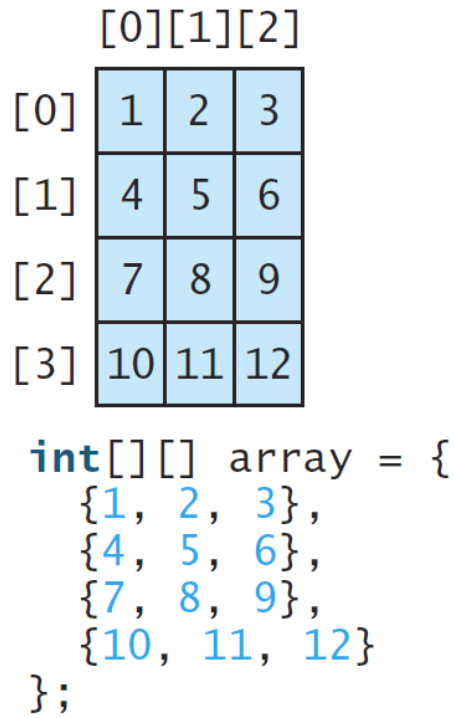
```
1 ...
2         int rowSize = 4; // row size
3         int colSize = 3; // column size
4         T[][] x = new T[rowSize][colSize];
5 ...
```




(a)



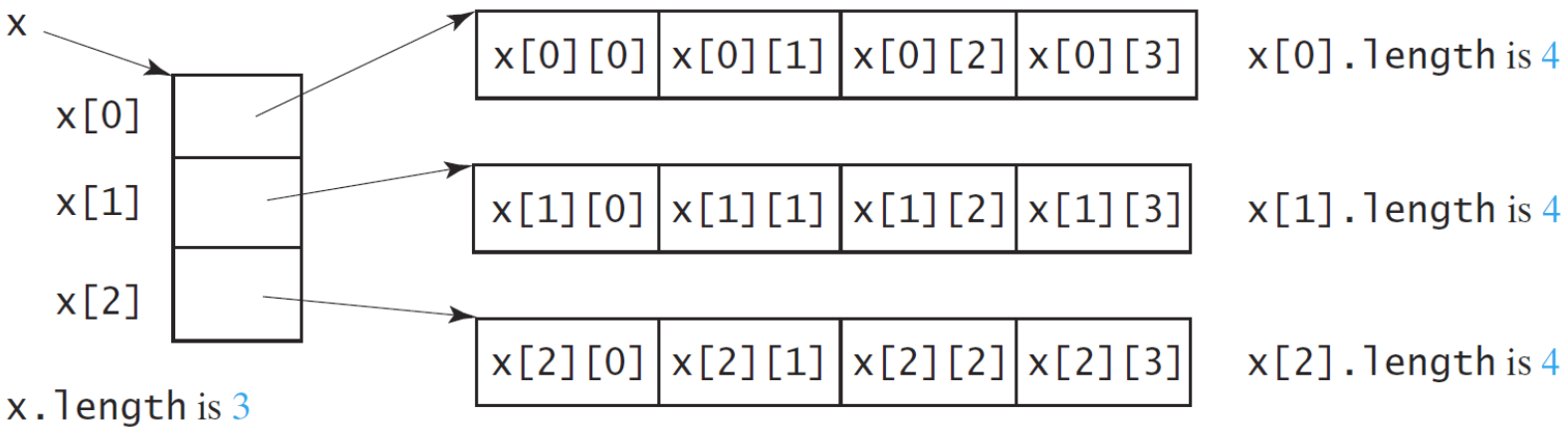

(b)



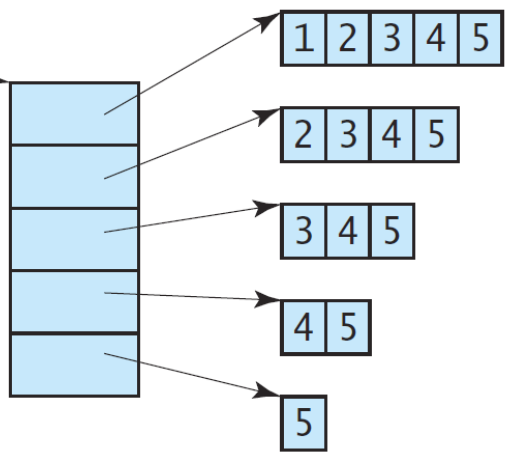

(c)

- Case (c) shows that we can create a 2D array by enumeration.

# Reality




```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



# Example[13]

```
...
        int[][] A = {{1, 2, 3}, {4, 5}, {6}};

        // conventional for loop
        for (int i = 0; i < A.length; i++) {
            for (int j = 0; j < A[i].length; j++)
                System.out.printf("%2d", A[i][j]);
            System.out.println();
        }

        // for-each loop
        for (int[] B: A) {
            for (int item: B)
                System.out.printf("%2d", item);
            System.out.println();
        }
...
```

[13] Thanks to a lively discussion on January 31, 2016.

# Exercise: Matrix Multiplication

Write a program which determines $C = A \times B$ for the input matrices $A_{m \times n}$ and $B_{n \times q}$ for $m, n, q \in \mathbb{N}$.

- You may use the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

  where $a_{ik}$, $i = 1, 2, \ldots, m$ is a shorthand for $A$ and $b_{kj}$, $j = 1, 2, \ldots, q$ for $B$.
- Time complexity: $O(n^3)$ (Why?)

```
1 class Lecture6 {
2
3                     "Methods"
4
5 }
6
7 // keywords:
8 return
```

# Methods[15]

- Methods can be used to define reusable code, and organize and simplify code.
- The idea of function originates from math, that is,

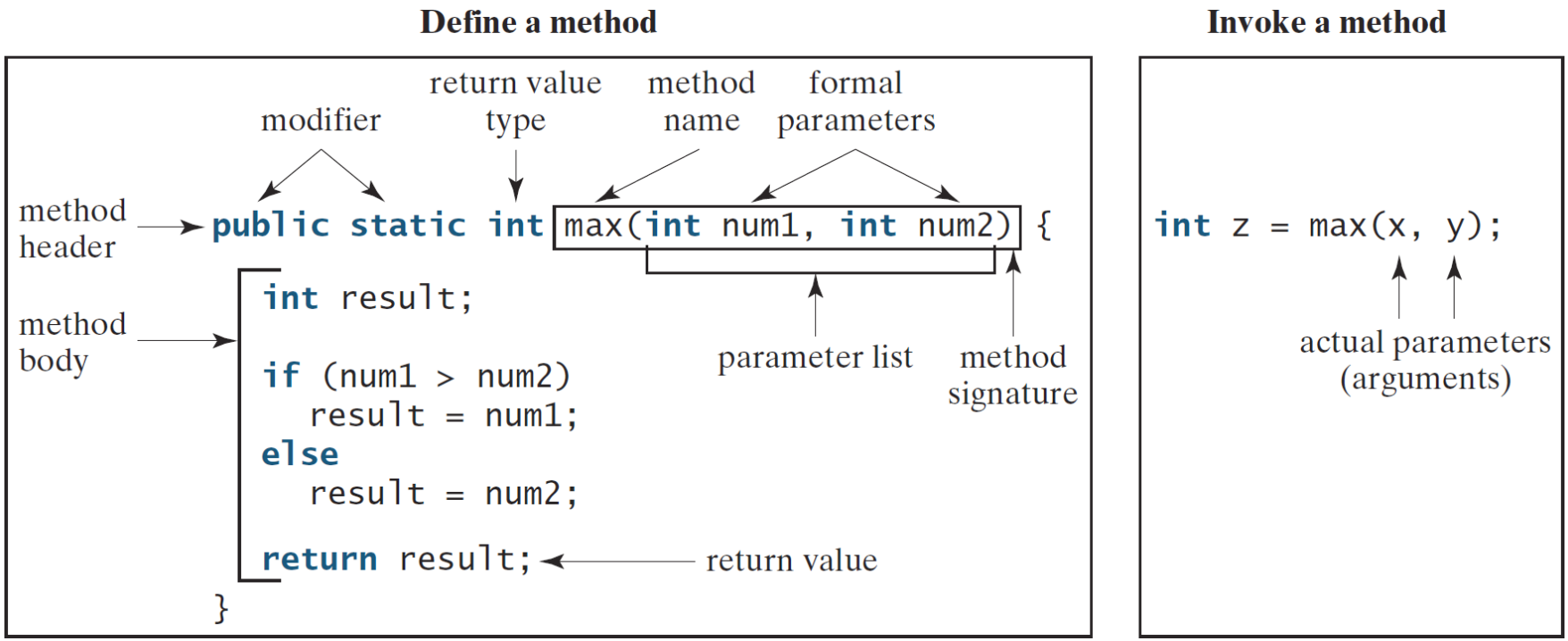
$$y = f(x),$$

  where $x$ is the input parameter[14] and $y$ is the function value.
- In computer science, each input parameter should be declared with a specific type, and a function should be assigned with a return type.

[14] Recall the multivariate functions. The input can be a vector, say the position vector $(x, y, z)$.

[15] Aka procedures and functions.

# Example: max


Define a method
Invoke a method
modifier
return value type
method name
formal parameters
method header
method body
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
parameter list
method signature
return value
int z = max(x, y);
actual parameters (arguments)

```
...
    modifier returnType methodName(listOfParameters) {
        // method body
    }
...
```

- The *modifier* could be static and public (for now).
- The *returnType* could be primitive types and reference types.
  - If the method does not return any value, then the return type is void.
- The *listOfParameters* is the input of the method, separated by commas if there are multiple items.
  - Note that a method could have no input.[16]
- The method name and the parameter list together are called the method signature.[17]

[16] For example, **Math.random()**.

[17] Method overloading depends this. We will see it soon.

# More Observations

- There are alternatives to the method **max**():

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

```
1 ...
2     public static int max(int x, int y) {
3         return x > y ? x : y;
4     }
5 ...
```

"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

# The return Statement

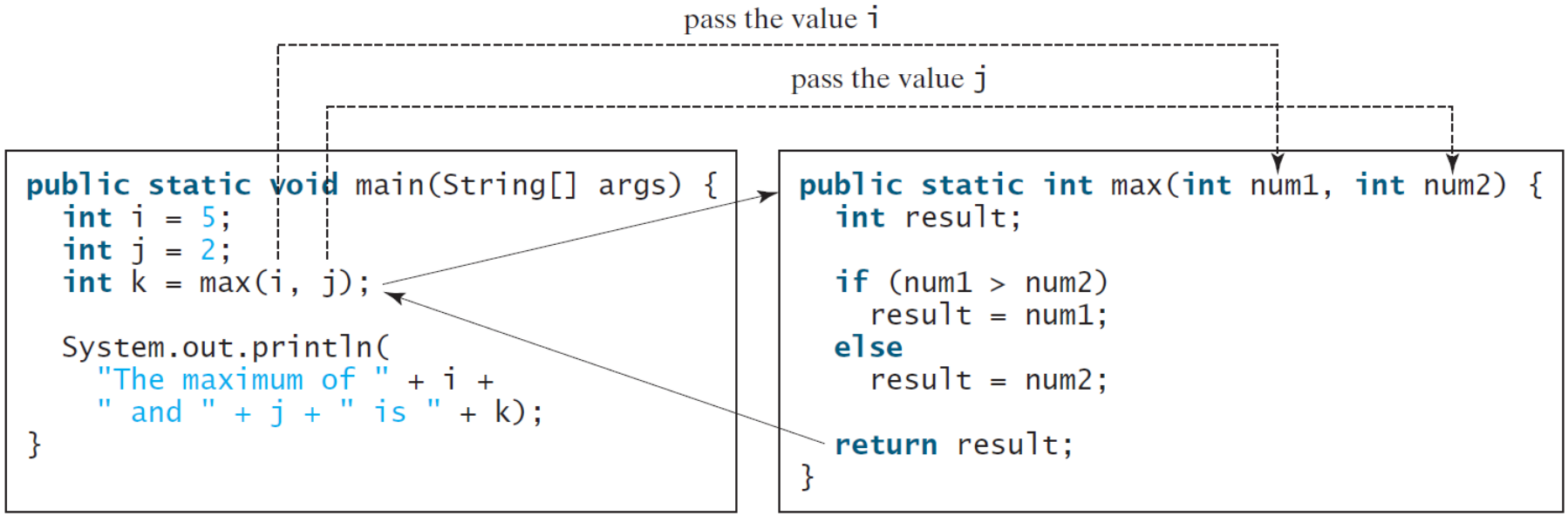
- The return statement is the end point of the method.
- A callee is a method invoked by a caller.
- The callee returns to the caller if the callee
  - completes all the statements (w/o a return statement, say **main**());
  - reaches a return statement;
  - throws an exception (introduced later).
- As you can see, the return statement is not necessarily at the bottom of the method.[18]
- Once one defines the return type (except void), the method should guarantee to return a value or an object of that type.

[18] Thanks to a lively discussion on November 22, 2015.

# Bad Exampls

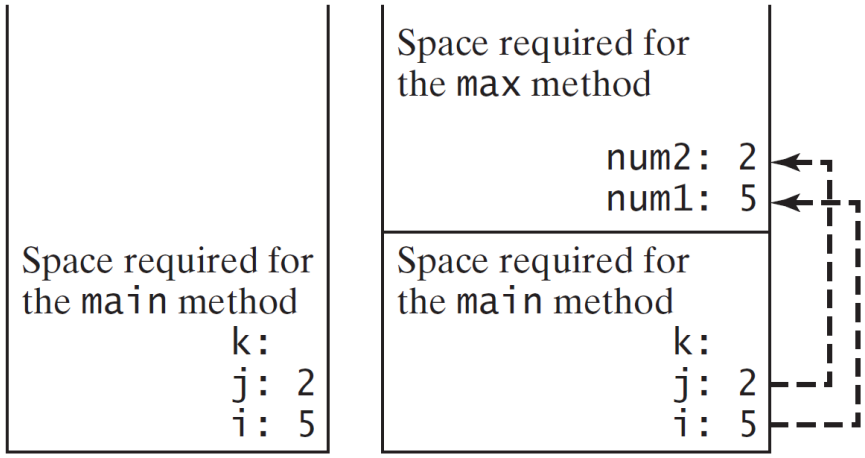
```
1  ...
2      public static int fun1() {
3          while (true);
4          return 0; // unreachable code
5      }
6
7      public static int fun2(int x) {
8          if (x > 0) {
9              return x;
10         }
11         // what if x < 0?
12     }
13 ...
```

# Method Invocation



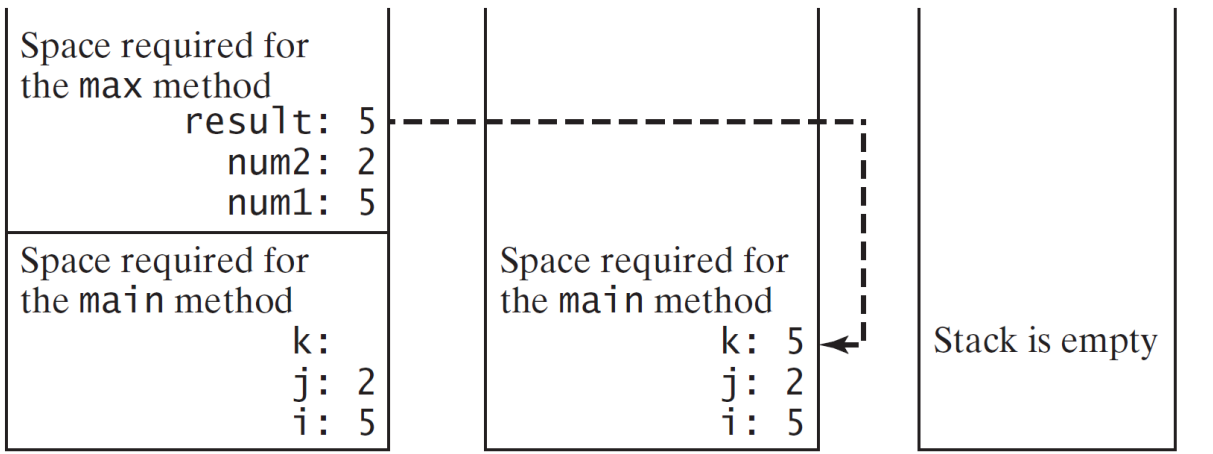

- Note that the input parameters are sort of variables declared within the method as placeholders.
- When calling the method, one needs to provide arguments, which must match the parameters in order, number, and compatible type, as defined in the method signature.

- 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 invocation of methods, OS creates a frame which stores necessary information, and the frame is pushed in the call stack.
- The callee transfers the program control back to the caller once the callee finishes its job.




(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.
- We cannot duplicate the variables whose names are identical in the same level.

# Example

```
1  public class ScopeDemo {
2
3      public static int x = 1; // class level, also called a field
4
5      public static void main(String[] args) {
6          System.out.println(x); // output 1
7          int x = 2; // method level, also called local variable
8          x++;
9          System.out.println(x); // output 3
10         addOne();
11         System.out.println(x); // output ?
12     }
13
14     public static void addOne() {
15         x = x + 1;
16         System.out.println(x); // output ?
17     }
18 }
```

# A Math Toolbox: **Math** Class

- The **Math** class provides basic mathematical functions and 2 global constants **Math.PI**[19] and **Math.E**[20].
- All methods are public and static.
  - For example, max, min, round, ceil, floor, abs, pow, exp, sqrt, cbrt, log, log10, sin, cos, asin, acos, and random.
- Full document for **Math** class can be found here.
- You are expected to read the document!

[19] The constant $\pi$ is a mathematical constant, the ratio of a circle's circumference to its diameter, commonly approximated as 3.141593.

[20] The constant $e$ is the base of the natural logarithm. It is approximately equal to 2.71828.

# Method Overloading

- Methods with the same name can coexist and be identified by the method signatures.

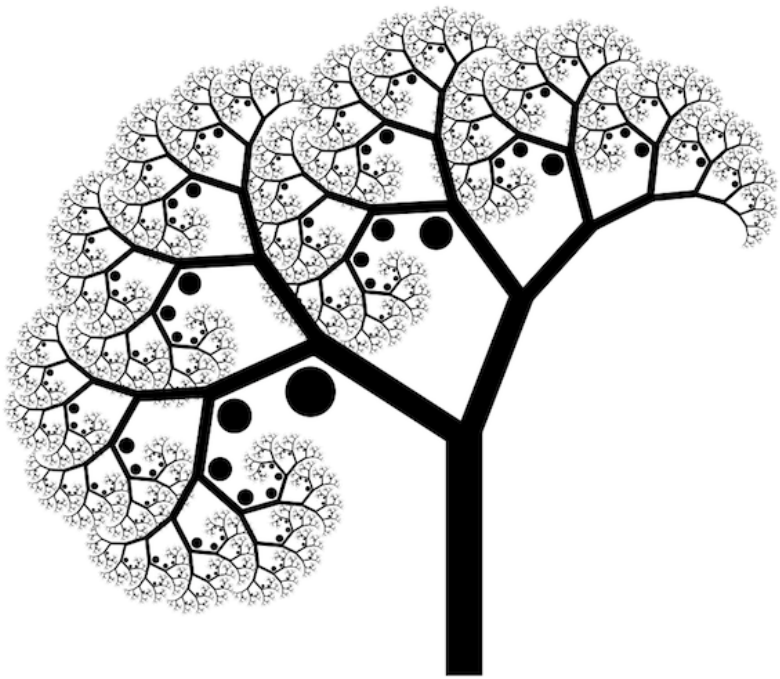
```
1 ...
2         public static int max(int x, int y) { ... }
3         // different numbers of inputs
4         public static int max(int x, int y, int z) { ... }
5          // different types
6         public static double max(double x, double y) { ... }
7 ...
```

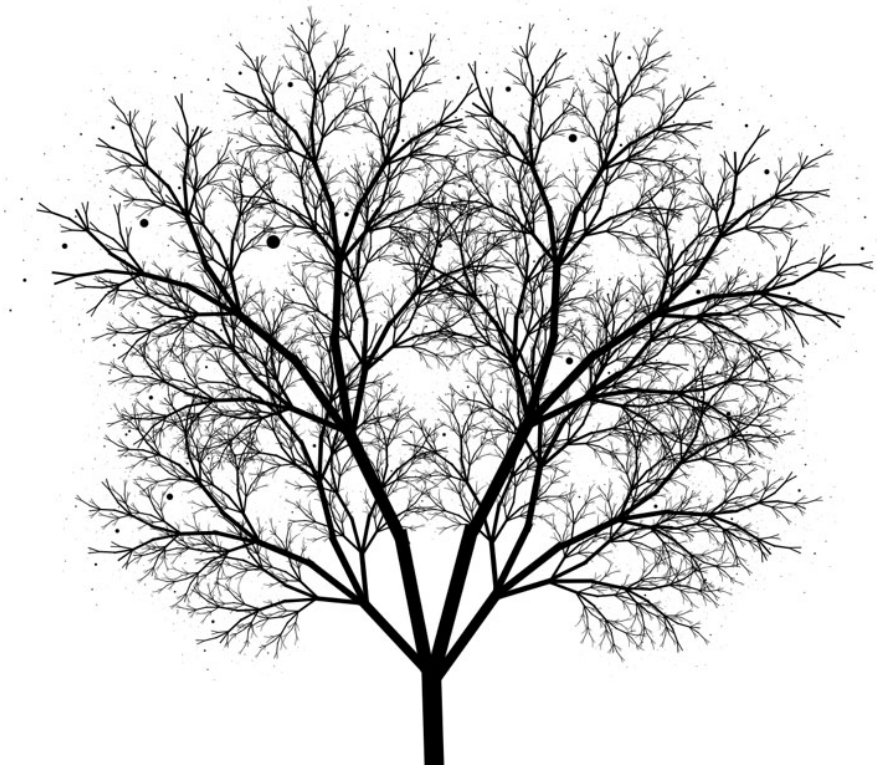
# Recursion[21]

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

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

[21] Recursion is a commom pattern in nature.





- Try Fractal.

# Example

The factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than and equal to n.

- Note that $0! = 1$.
- For example,

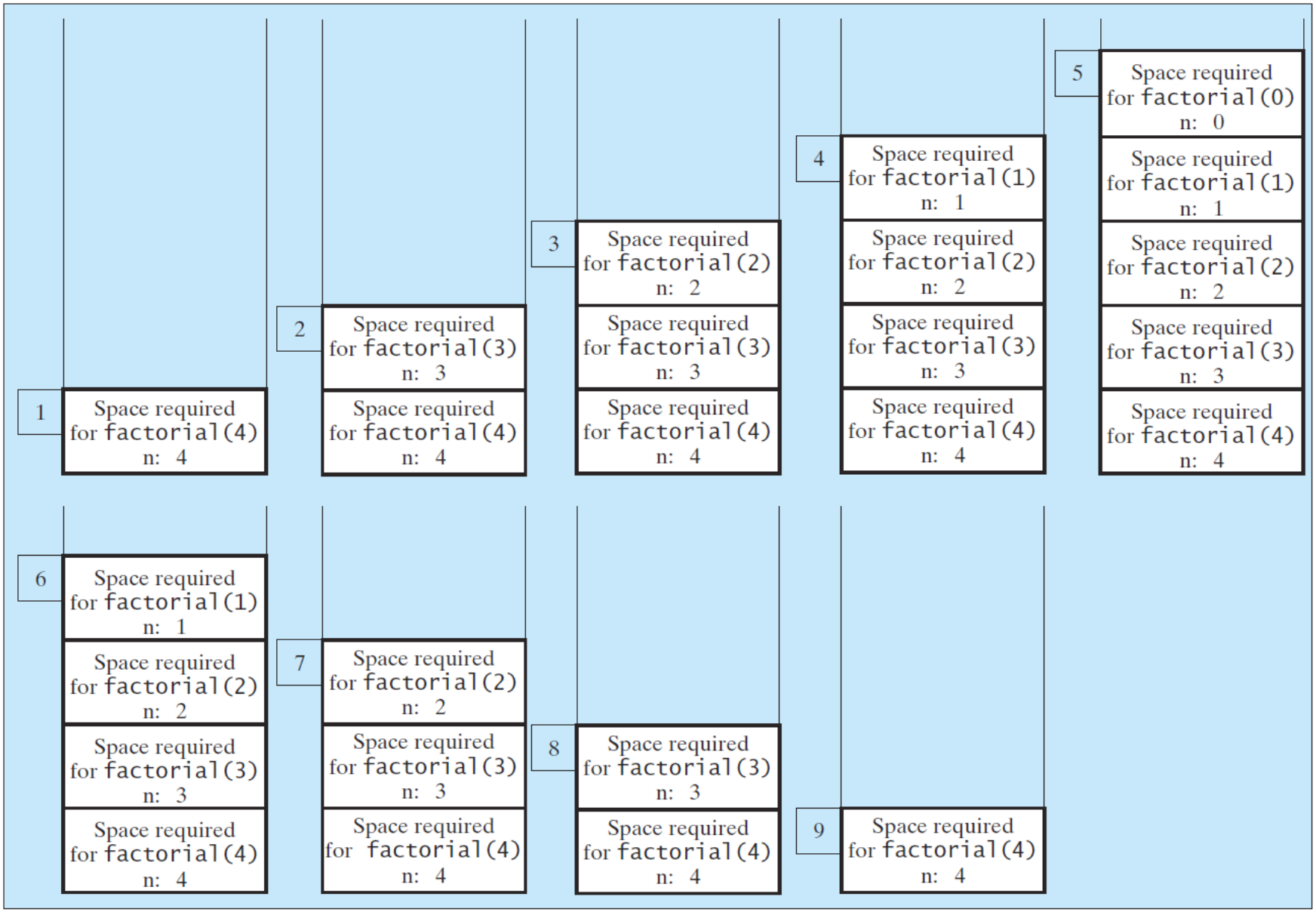
$$\begin{aligned} 4! &= 4 \times 3 \times 2 \times 1 \\ &= 4 \times 3! \\ &= 24. \end{aligned}$$

- Can you find the pattern?
  - $n! = n \times (n-1)!$
  - In general, $f(n) = n \times f(n-1)$.

Write a program which determines $n!$.

```
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 ...
```

- Note that there must be a base case in recursion.
- Time complexity: $O(n)$
- Can you implement the same method by using a loop?


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

# Equivalence: Loop Version

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

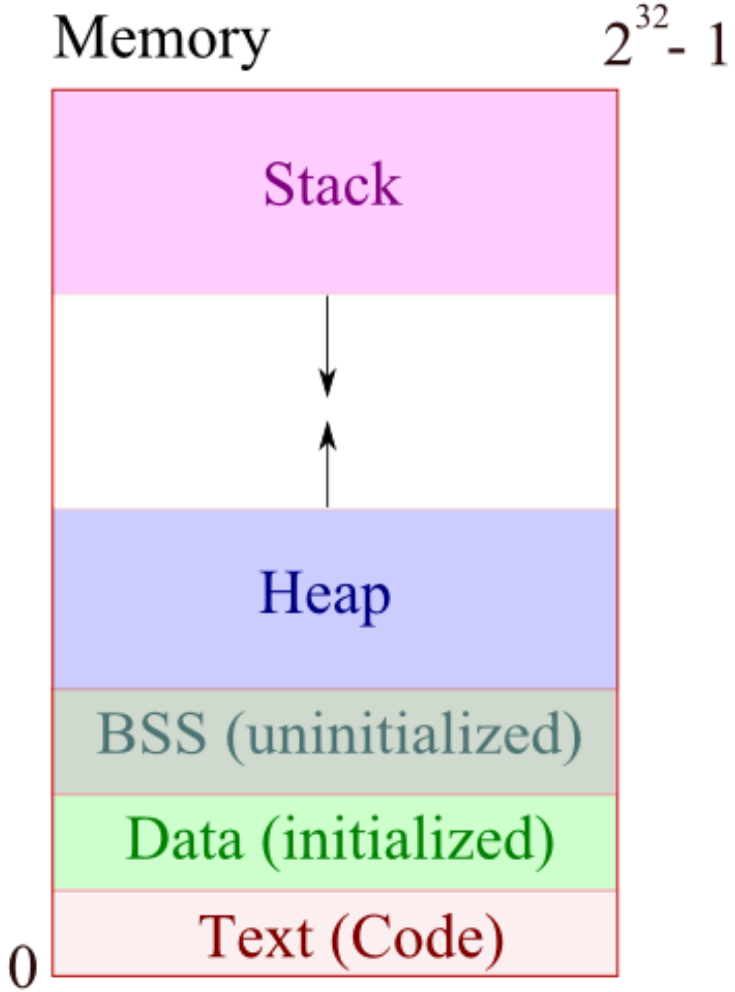
- Time complexity: $O(n)$
- One intriguing question is, Can we always turn a recursive method into a loop version of that?
- Yes, theoretically.[22]

[22] The Church-Turing Thesis proves it if the memory serves.

# Remarks

- Recursion bears substantial overhead.
- So the recursive algorithm may execute a bit more slowly than the iterative equivalent.
- Additionally, a deeply recursive method depletes the call stack, which is limited, and causes stack overflow soon.

# Memory Layout



# Example: Fibonacci Numbers

Write a program which determines $F_n$, the $(n+1)$-th Fibonacci number.

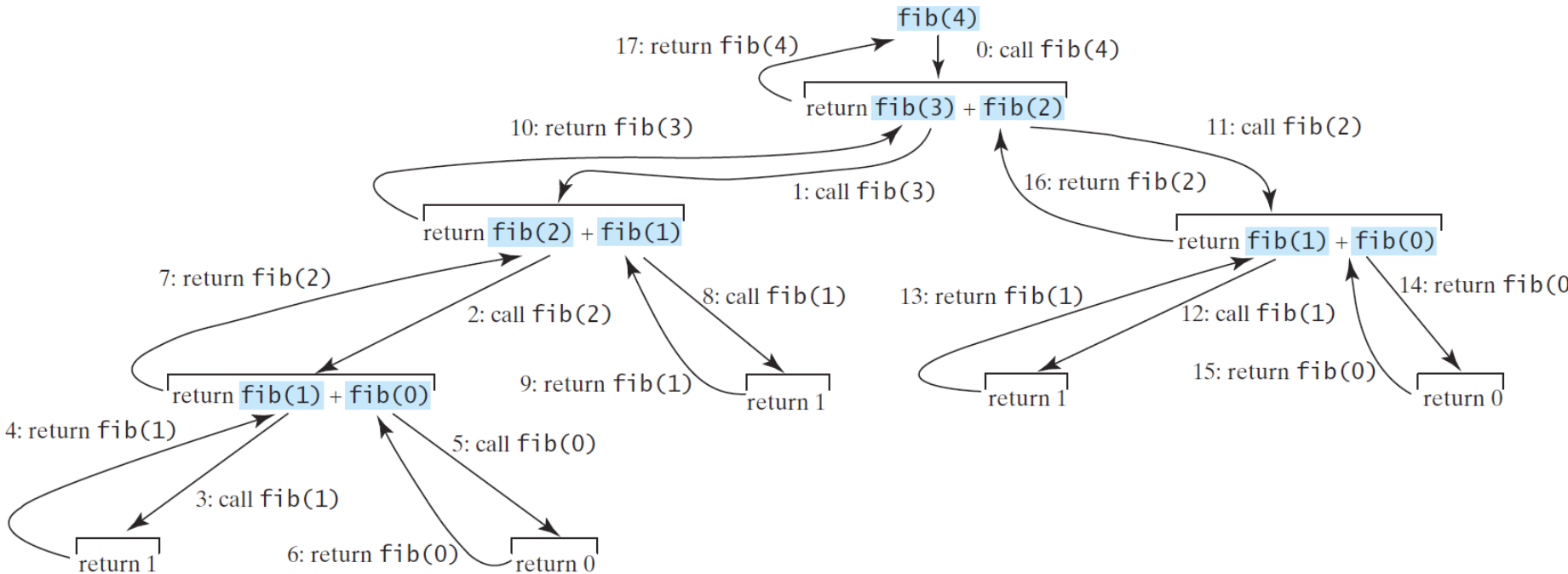
- The first 10 Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21,$ and 34.
- The sequence of Fibonacci numbers can be defined by the recurrence relation

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

where $n \geq 2$ and $F_0 = 0, F_1 = 1$.

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

- This recursive implementation is straightforward.
- Yet, this algorithm isn't efficient since it requires more time and memory.
- 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)
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

```
...
    public static double fibIter(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.
- It implies that the recursive one is not optimal.
- Could you find a linear recursion for Fibonacci numbers?
- You may try more examples.[23]

[23] See `http://introcs.cs.princeton.edu/java/23recursion/`.

# Divide and Conquer

- For program development, we use the divide-and-conquer strategy[24] to decompose the original problem into subproblems, which are more manageable.
  - For example, selection sort.
- Pros: easier to write, reuse, debug, modify, maintain, and also better facilitating teamwork

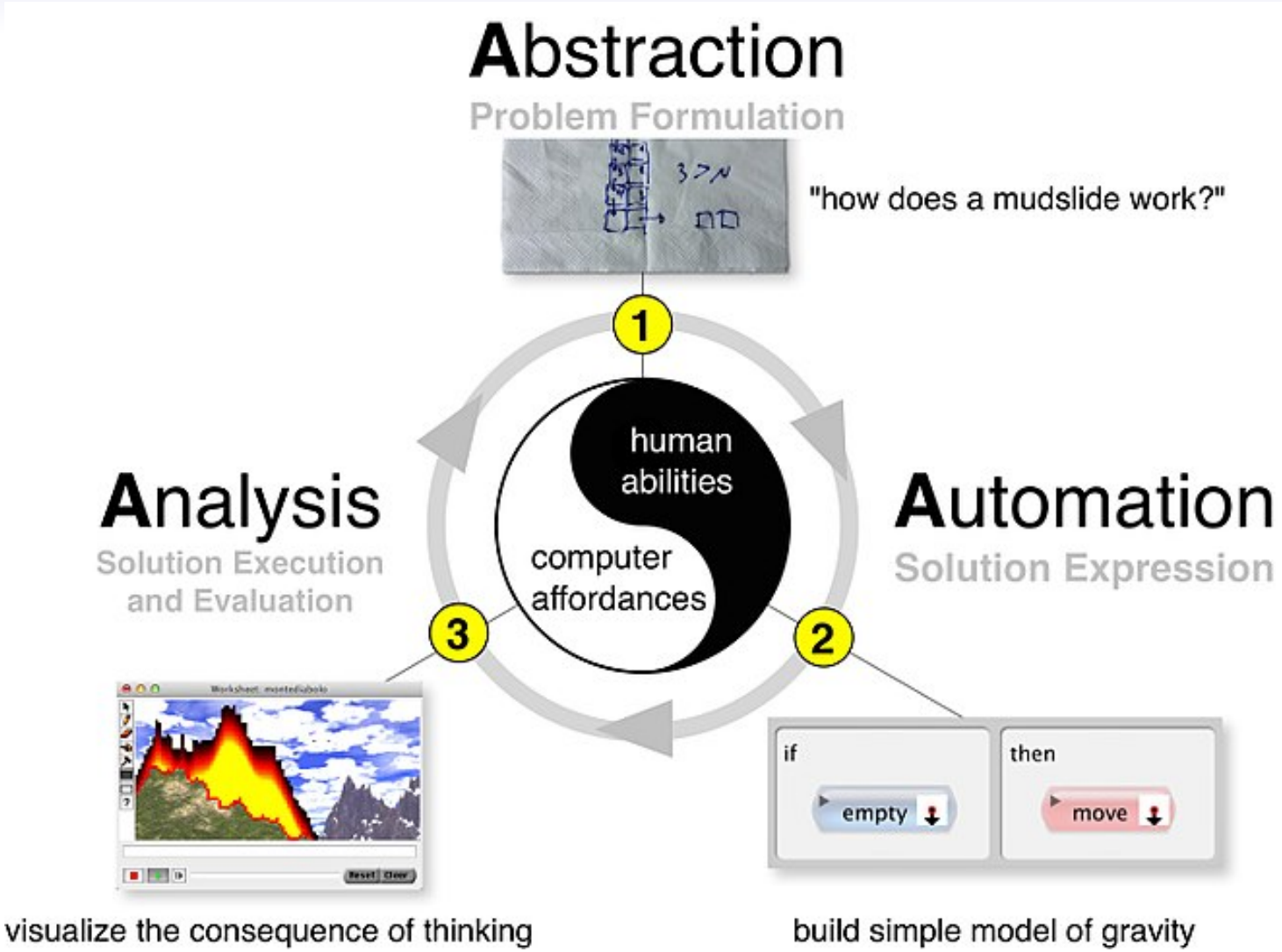
[24] Aka stepwise refinement.

# Computational Thinking

- Computational thinking is taking an approach to solving problems, designing systems and understanding human behavior that draws on concepts fundamental to computing.[25]
  - solve problems: mathematical thinking
  - design systems: engineering thinking[26]
  - understand human behavior: scientific thinking

[25] Read `http://rsta.royalsocietypublishing.org/content/366/1881/3717.full`.

[26] Design and evaluate a large and complex system that operates within the constraints of the real world.

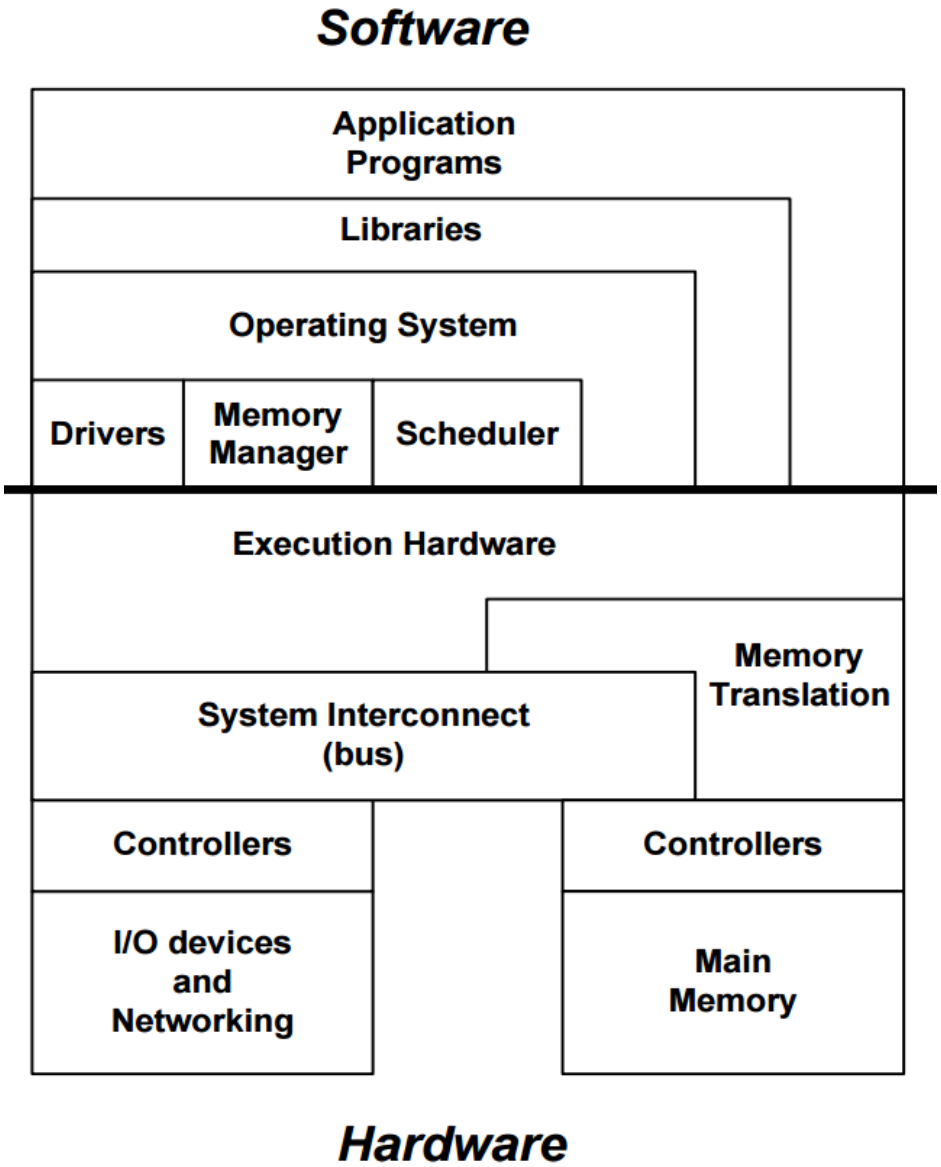



https://en.wikipedia.org/wiki/File:The_Computational_Thinking_Process.jpg

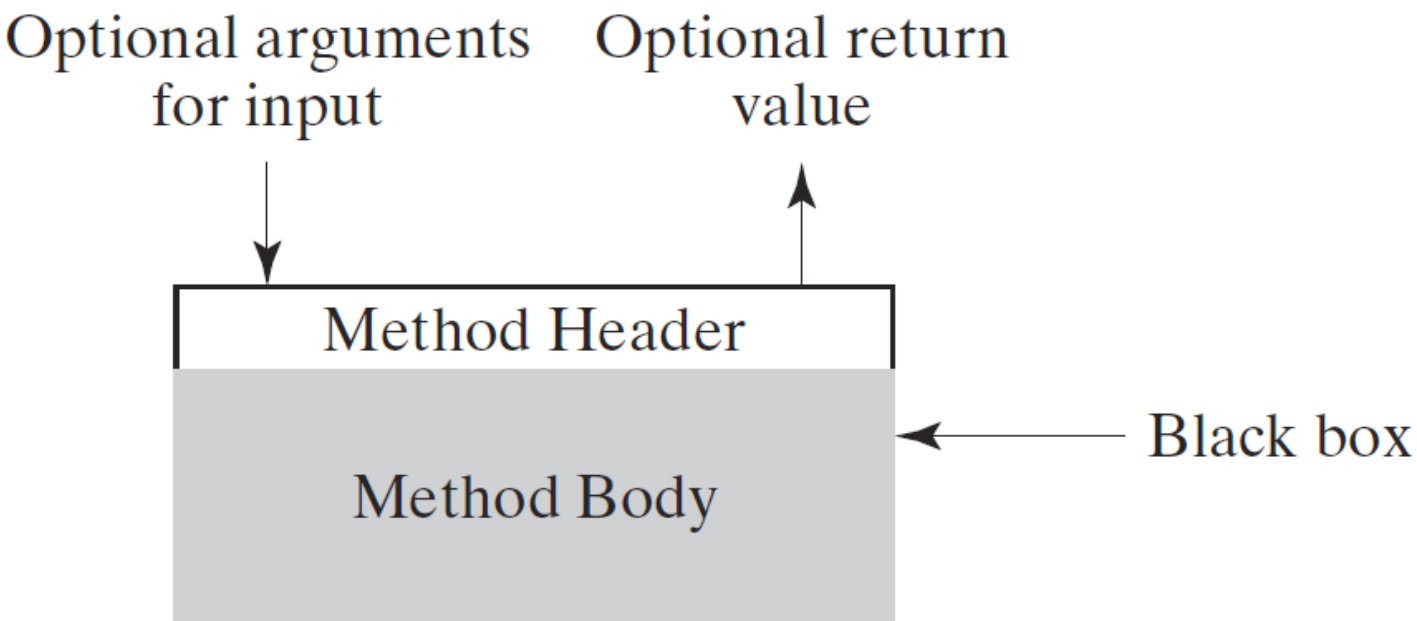
# Computational Thinking Everywhere

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

# Example: Abstraction of Computer System



# Example: Methods as Control Abstraction



# Abstraction (Concluded)

- Control abstraction is the abstraction of actions while data abstraction is that of data structures.
- One can view the notion of an object as a way to combine abstractions of data and actions.