ADT and Problem Solving
Lecture 3-4

Array of Object + Sorting + Iterative vs. Resursive

Pree Thiengburanathum, PhD.

Agenda

- Array of Object
- Basic Sorting Algorithms

Array of Object in Java

- An array is a container object that holds a fixed number of values of a single type.
- When you combine arrays with objects, you can create arrays that hold objects instead of primitive types like int or char.
- This allows you to manage collections of objects efficiently and perform operations on them collectively.

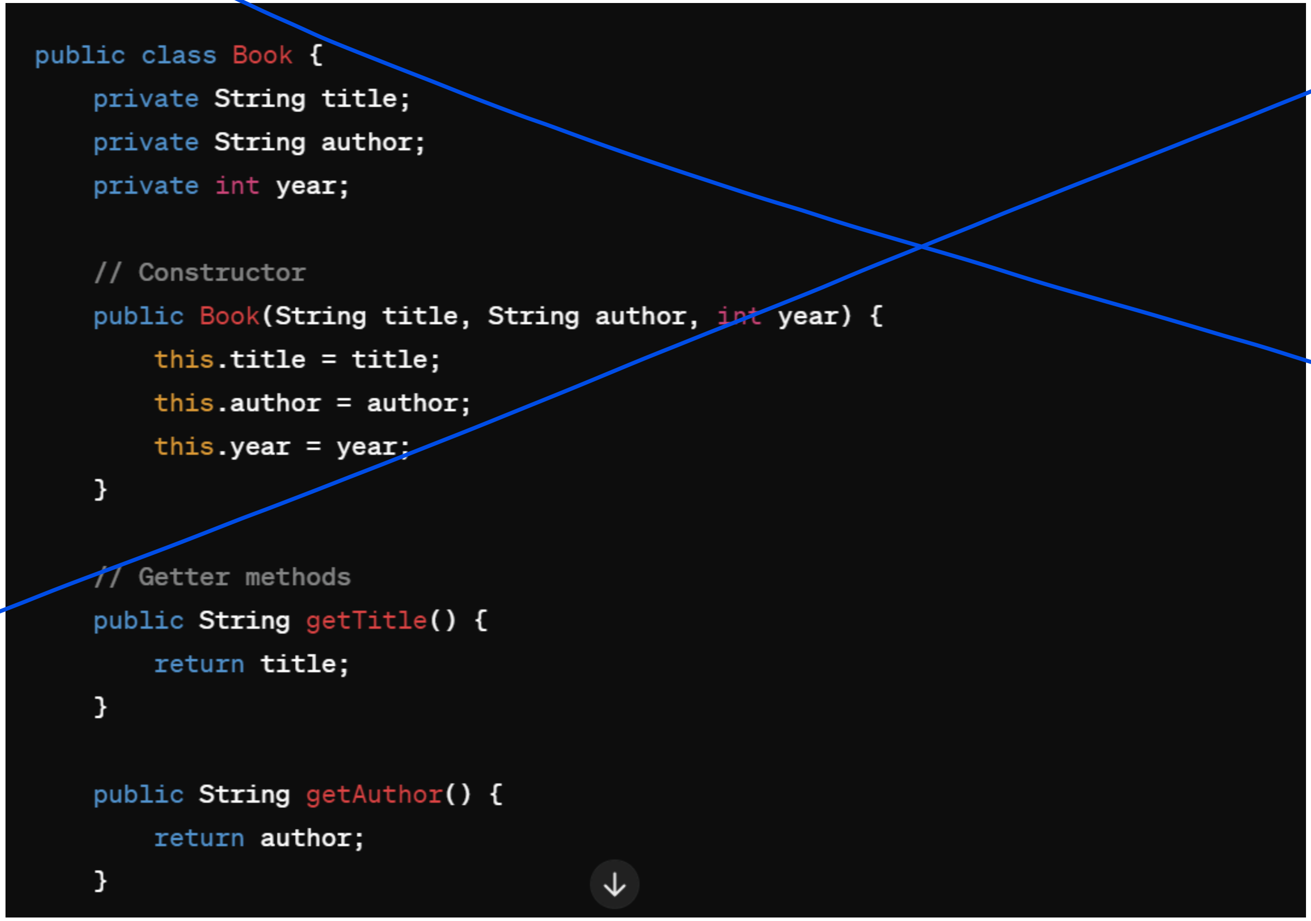
Array of object in Java

- Arrays of objects are fundamental in Java for managing multiple objects and are widely used in various applications.
- From simple data lists to complex business logic.
- They offer a simple way to store and manipulate groups of objects with similar properties.

Steps

1. Define a class
2. Create an Array of Objects
3. Initialize the Array
4. Use the Array

Code example (Define a class)



```java
public class Book {
    private String title;
    private String author;
    private int year;

    // Constructor
    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    // Getter methods
    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }
```

Code example (Create an Array of Object and initialize the array)

- Book[] myBooks = new Book[5]; // Creates an array to hold 5 Book objects

- myBooks[0] = new Book("1984", "George Orwell", 1949);
- myBooks[1] = new Book("To Kill a Mockingbird", "Harper Lee", 1960);
- myBooks[2] = new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925);
- myBooks[3] = new Book("Pride and Prejudice", "Jane Austen", 1813);
- myBooks[4] = new Book("Catch-22", "Joseph Heller", 1961)

Code example (Use the array)

```
for (int i = 0; i < myBooks.length; i++) {
    Book book = myBooks[i];
   System.out.println("Book " + (i+1) + ": " + book.getTitle() + " by " +
book.getAuthor() + " (" + book.getYear() + ")");
}
```

Benefit

- **Organization:** Arrays help keep your objects organized and accessible through index-based access.
- **Batch Operations:** You can perform operations like sorting, filtering, and searching on arrays of objects.
- **Memory Efficiency:** Having a single array holding references to objects can be more memory efficient than having separate variables for each object.

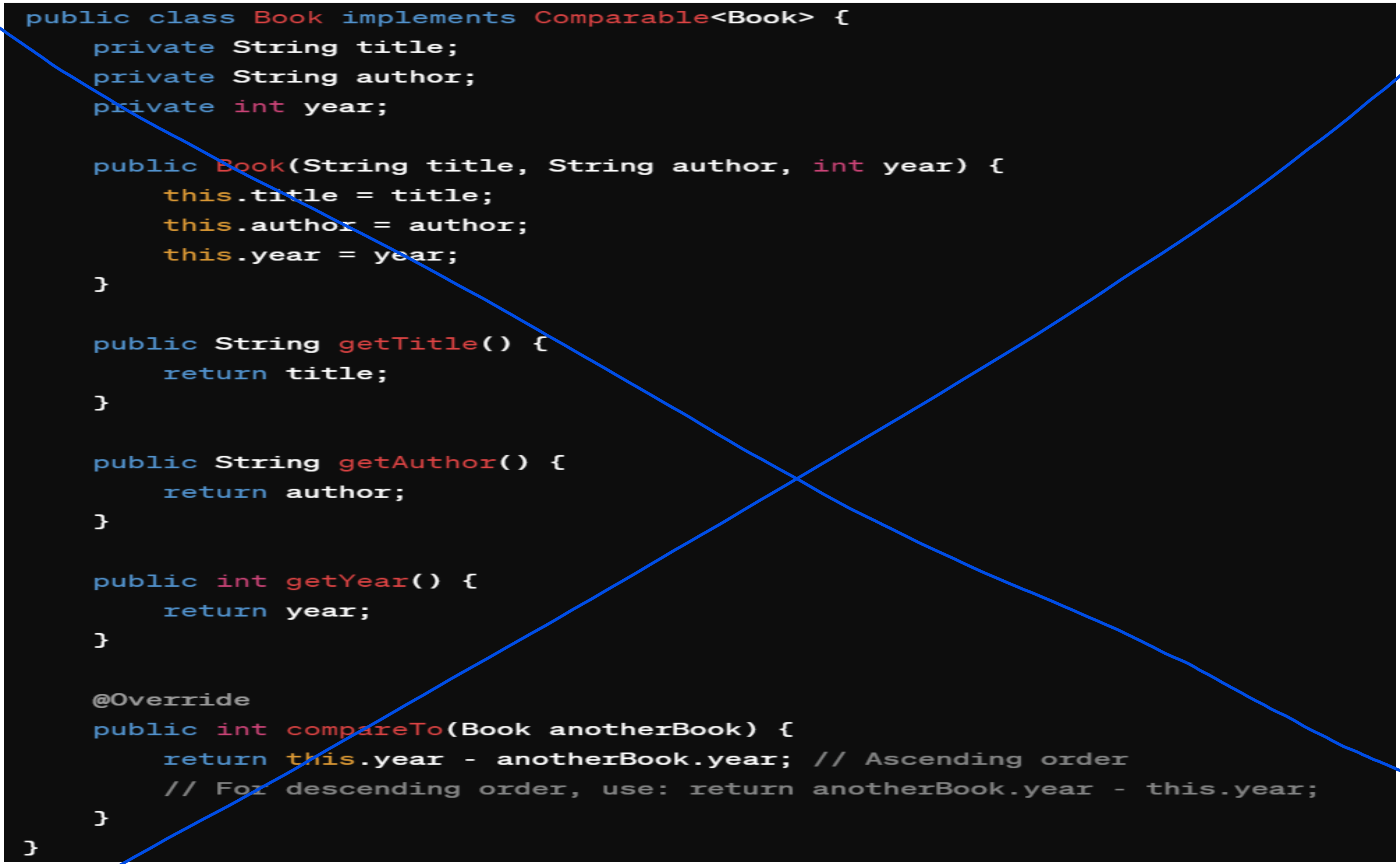
Object comparable

- In Java, the Comparable interface is used to impose a natural ordering on the objects of each class that implements it.
- This interface consists of a single method, *compareTo(),* that you need to override in your class to define the natural order.
- Useful when sorting arrays or collections of objects, as it provides a way for Java to understand how to order instances of your class.

Object comparable

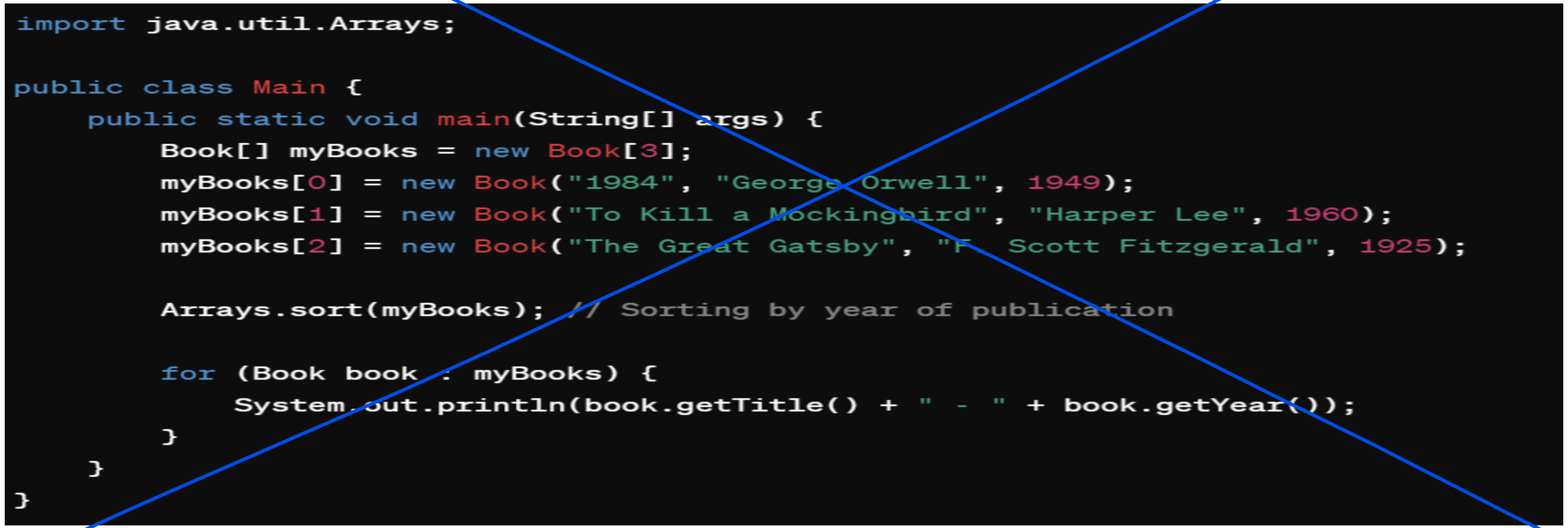
- Located at java.lang package, which means it doesn't require an import statement.
- The compareTo() method compares the current object with the specified object for order. It returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Example of code



```
public class Book implements Comparable<Book> {
    private String title;
    private String author;
    private int year;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public int getYear() {
        return year;
    }

    @Override
    public int compareTo(Book anotherBook) {
        return this.year - anotherBook.year; // Ascending order
        // For descending order, use: return anotherBook.year - this.year;
    }
}
```

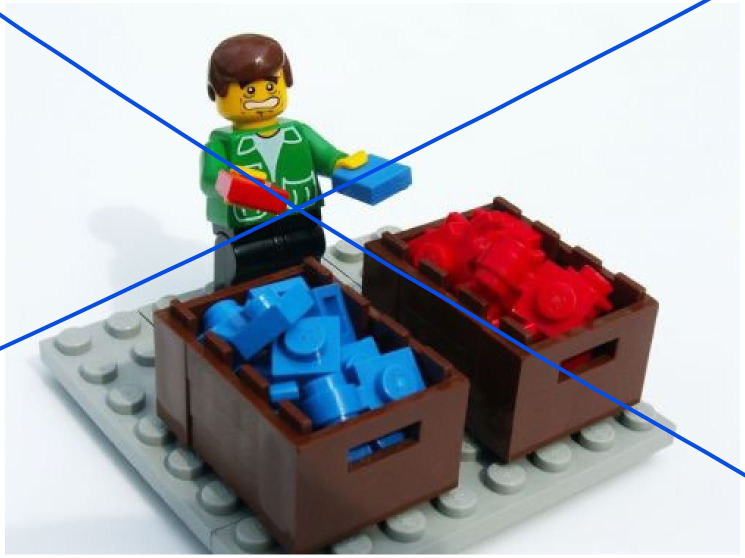
Example of code



```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Book[] myBooks = new Book[3];
        myBooks[0] = new Book("1984", "George Orwell", 1949);
        myBooks[1] = new Book("To Kill a Mockingbird", "Harper Lee", 1960);
        myBooks[2] = new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925);

        Arrays.sort(myBooks); // Sorting by year of publication

        for (Book book : myBooks) {
            System.out.println(book.getTitle() + " - " + book.getYear());
        }
    }
}
```

Sorting

- Arranging in an ordered sequence is called "sorting". Sorting is a common operation in many applications, and efficient algorithms to perform it have been developed.



Common sorting algorithms

- Bubble sort- exchange two adjacent elements if they are out of order. Repeat until array is sorted
- Insertion sort- scan successive elements for an out-of-order item, then insert the item in the proper place.
- Selection sort- find the smallest element in the array, and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted.

Common sorting algorithms

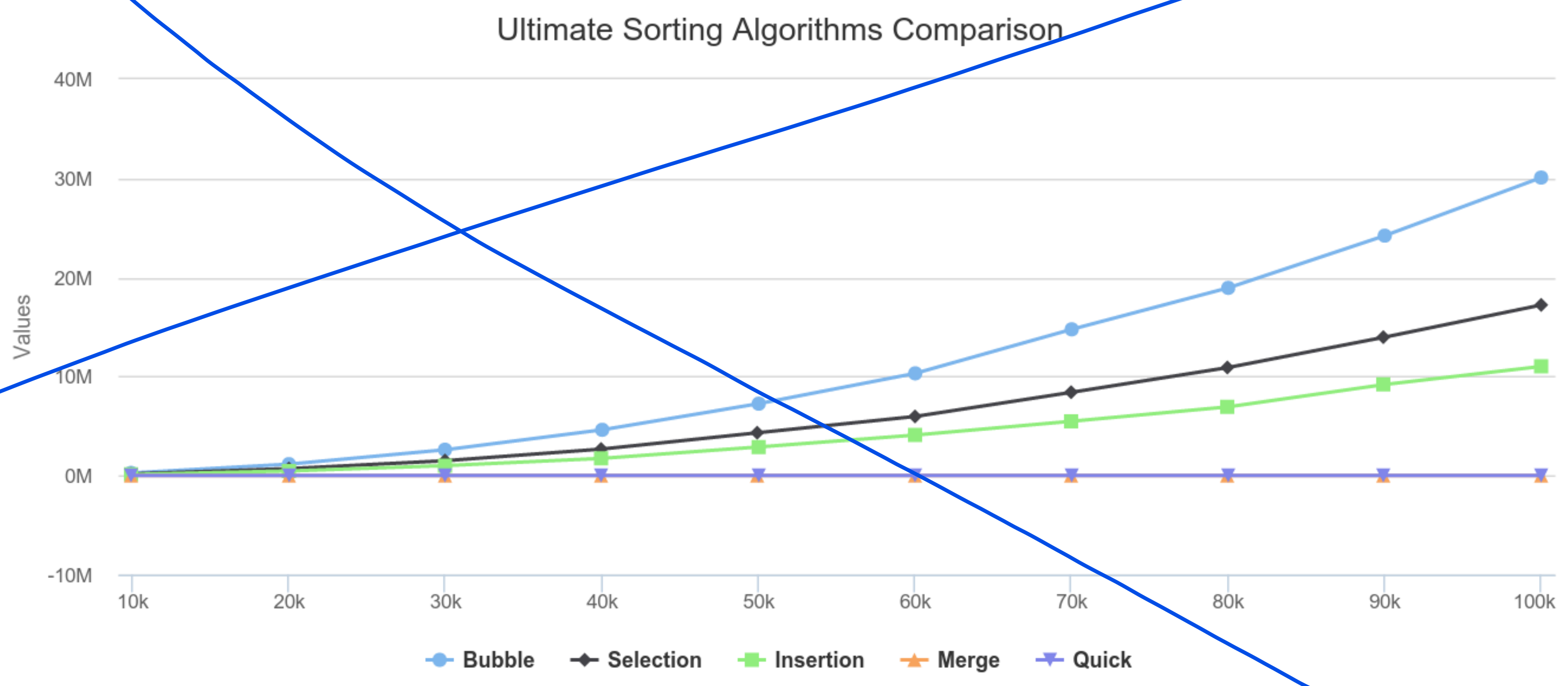
- Quick sort

Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively.

- Merge sort

Divide the list of elements in two parts, sort the two parts individually and then merge it.

Sorting Algorithms Comparison



Selection Sort

12 6	12 6	22	14	8	17

6	12 8	22	14	12 8	17

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Selection Sort

6	8	22	14	12	17

6	8	12	14	17	22

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Selection Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

- How efficient is selection sort?
  - In general, given n numbers to sort, it performs n^2 comparisons
- Why might selection sort be a good choice?
  - Simple to write code
  - Intuitive

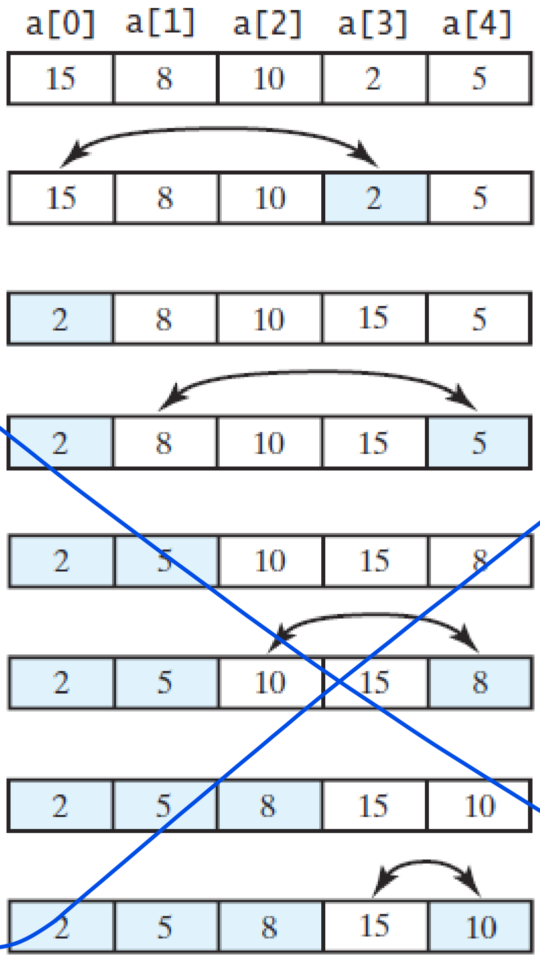
Selection Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

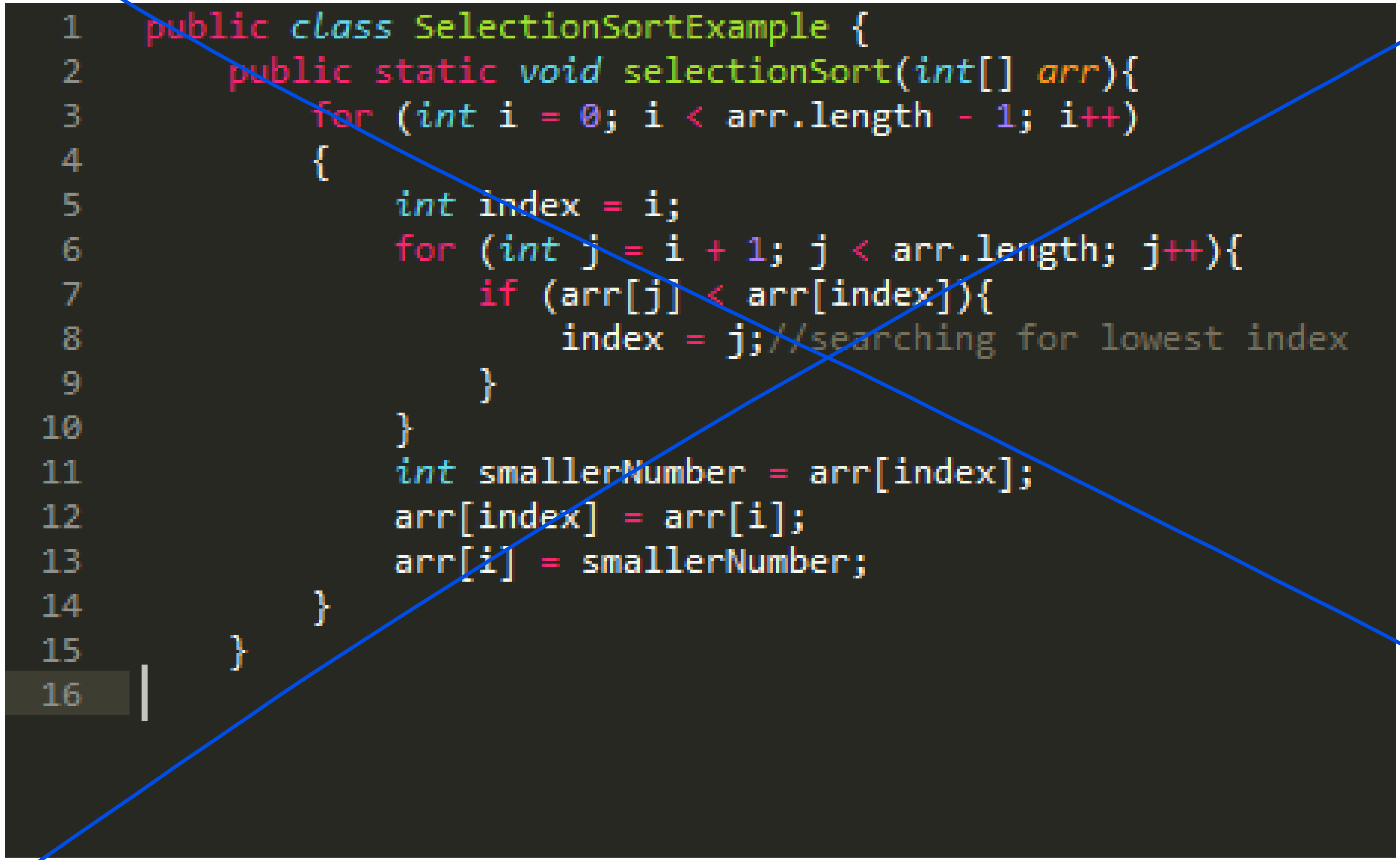
Try one!

15	3	11	19	4	7

A Selection sort

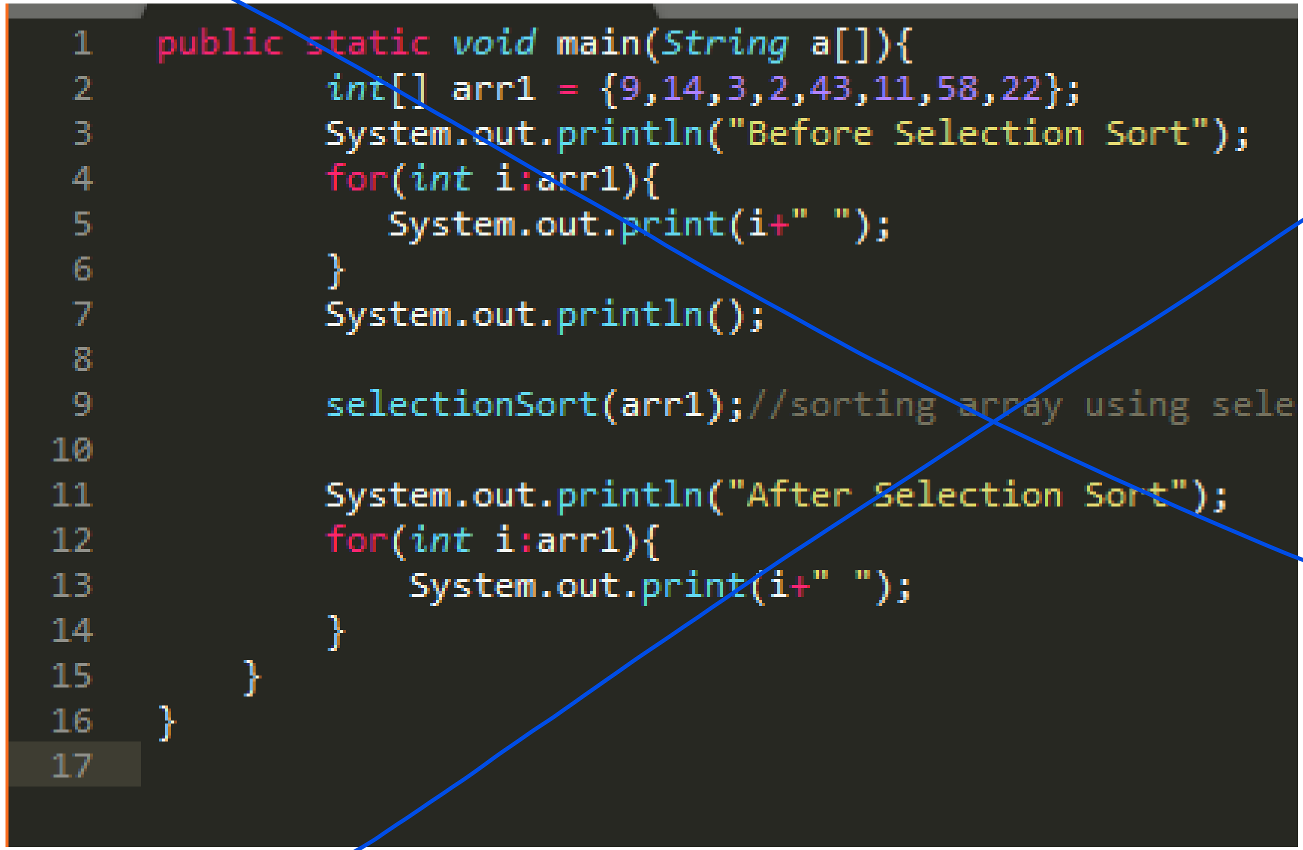


A Class for sorting an array using selection sort



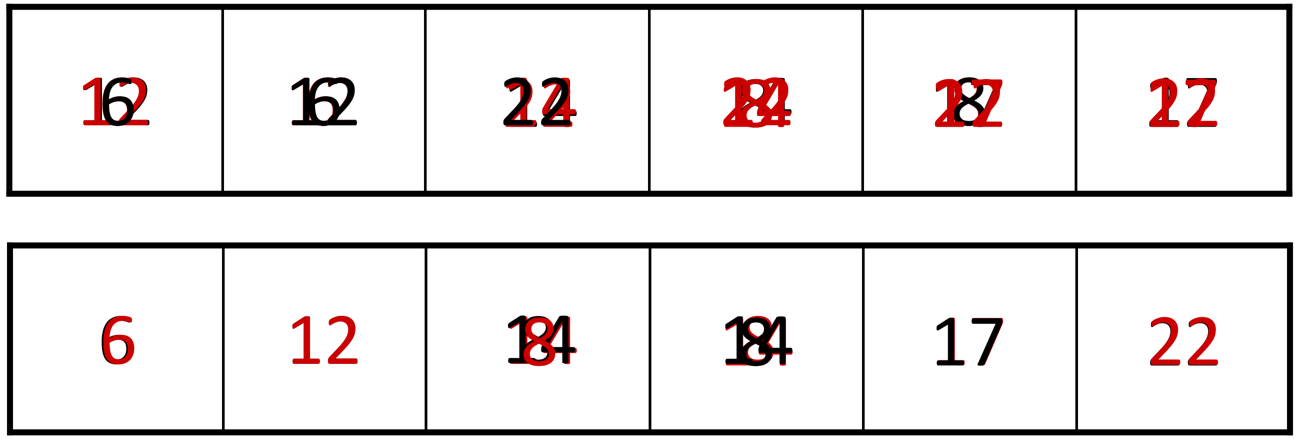
```java
public class SelectionSortExample {
    public static void selectionSort(int[] arr){
        for (int i = 0; i < arr.length - 1; i++)
        {
            int index = i;
            for (int j = i + 1; j < arr.length; j++){
                if (arr[j] < arr[index]){
                    index = j;//searching for lowest index
                }
            }
            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
    }
```

Calling the selection sort class



```
public static void main(String a[]){
        int[] arr1 = {9,14,3,2,43,11,58,22};
        System.out.println("Before Selection Sort");
        for(int i:arr1){
           System.out.print(i+" ");
        }
        System.out.println();

        selectionSort(arr1);//sorting array using sele

        System.out.println("After Selection Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
    }
}
```

Bubble Sort



- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

6	12 8	12 8	14	17	22

6	8	12	14	17	22

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them

---

- How efficient is bubble sort?
  - In general, given n numbers to sort, it performs n^2 comparisons
  - The same as selection sort

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
    - For each pair of adjacent numbers:
        - If the number on the left is greater than the number on the right, swap them

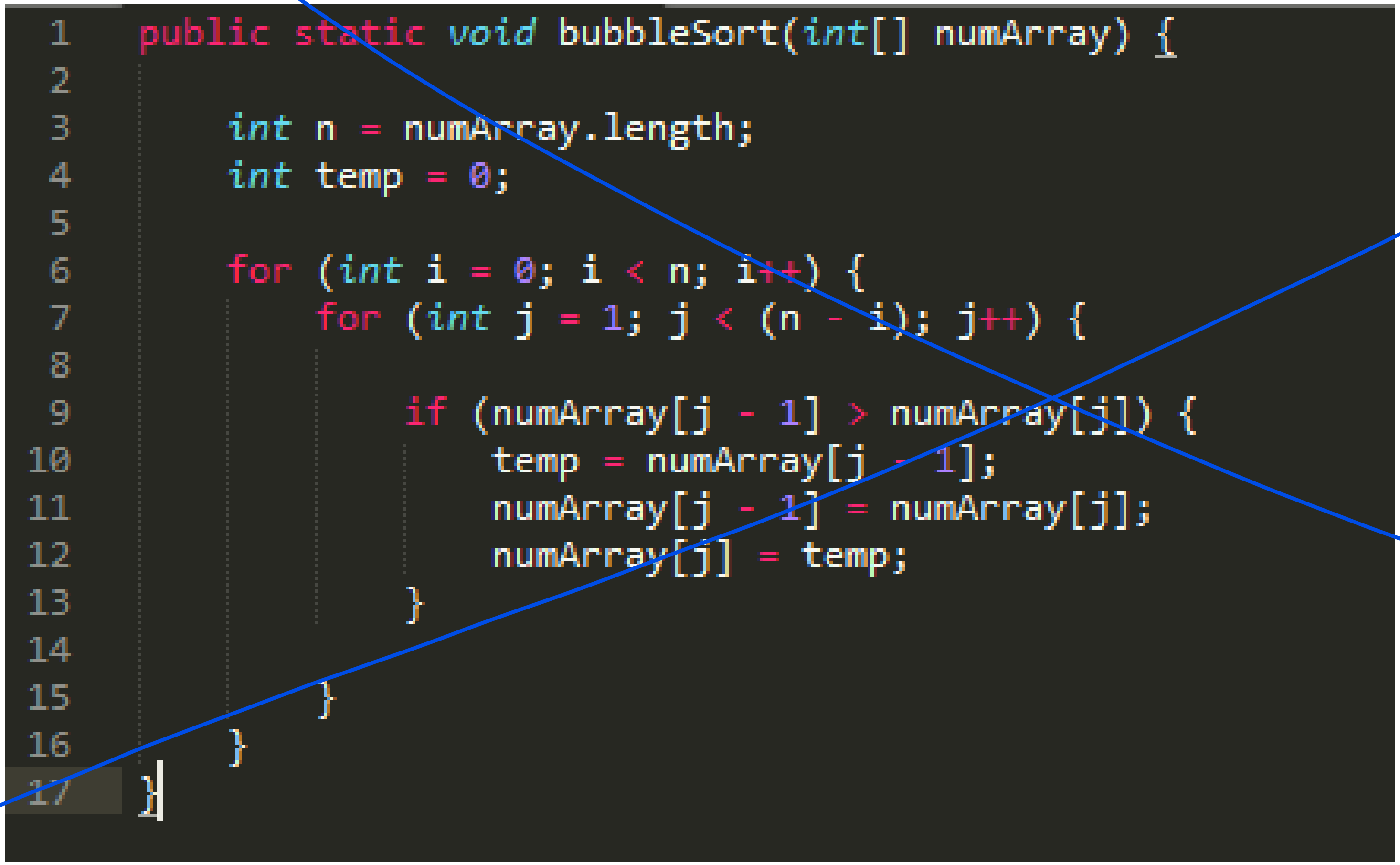
Try one!

1	23	2	56	9	8	10	100

Bubble Sort

```
1    1  23  2   56   9    8   10   100
2    1  2   23  56   9    8   10   100
3    1  2   23  9    56   8   10   100
4    1  2   23  9    8    56  10   100
5    1  2   23  9    8    10  56   100
---- finish the first traversal  ----
1    1  2   23  9    8    10  56   100
2    1  2   9   23   8    10  56   100
3    1  2   9   8    23   10  56   100
4    1  2   9   8    10   23  56   100
---- finish the second traversal ----
                    …
```

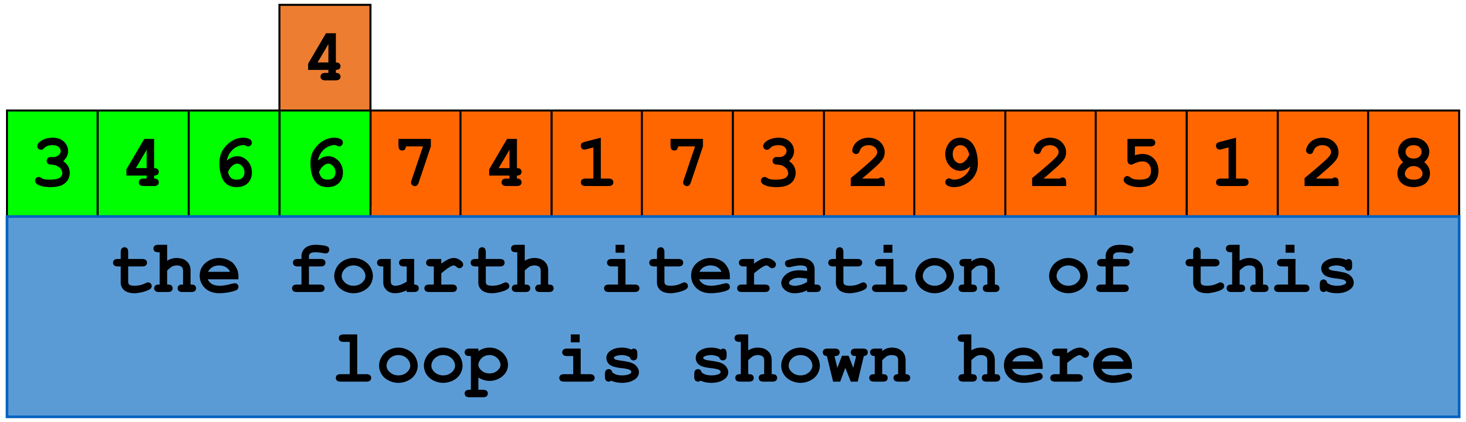
~~Java bubble sort example~~



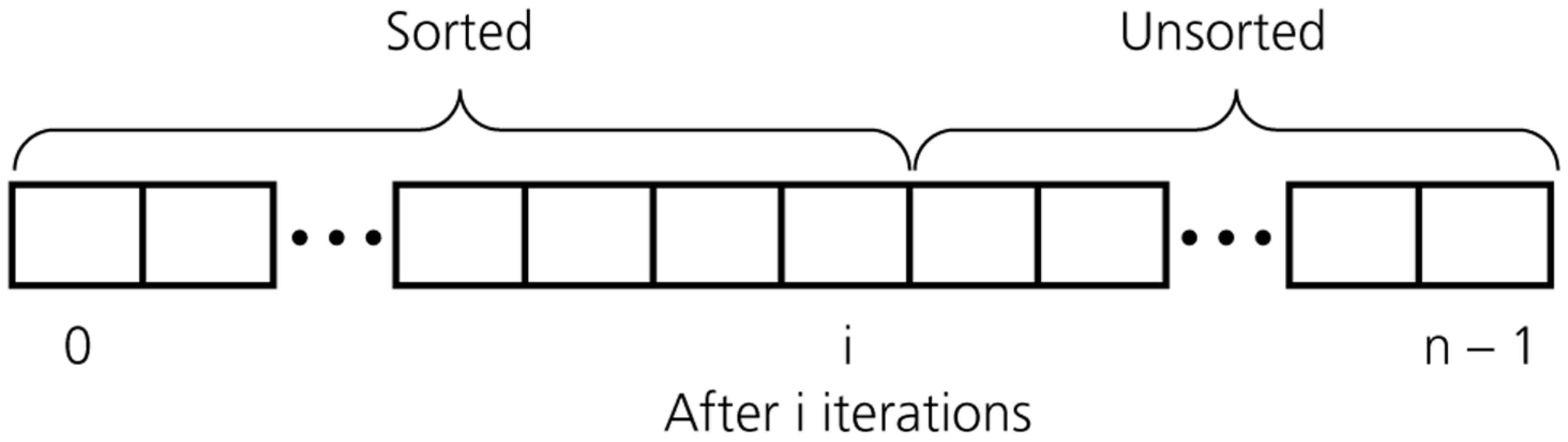
```
~~public static void bubbleSort(int[] numArray) {

    int n = numArray.length;
    int temp = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 1; j < (n - i); j++) {

            if (numArray[j - 1] > numArray[j]) {
                temp = numArray[j - 1];
                numArray[j - 1] = numArray[j];
                numArray[j] = temp;
            }

        }
    }
}~~
```

Insertion Sort

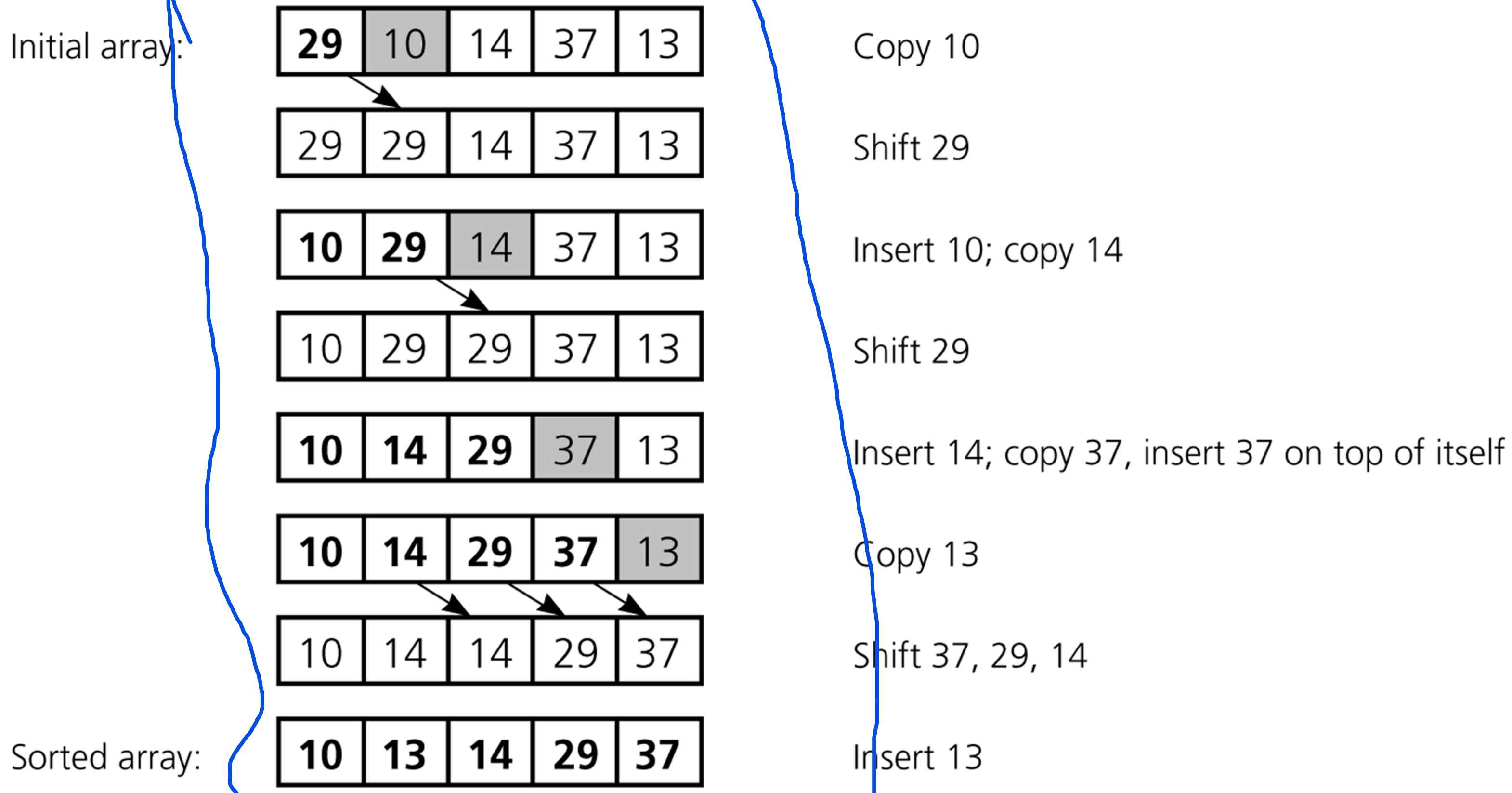
- while some elements unsorted:
  - Using linear search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
  - Move all the elements after the insertion location up one position to make space for the new element



An insertion sort partitions the array into two regions



An insertion sort of an array of five integers



Insertion sort

How efficient is bubble sort?

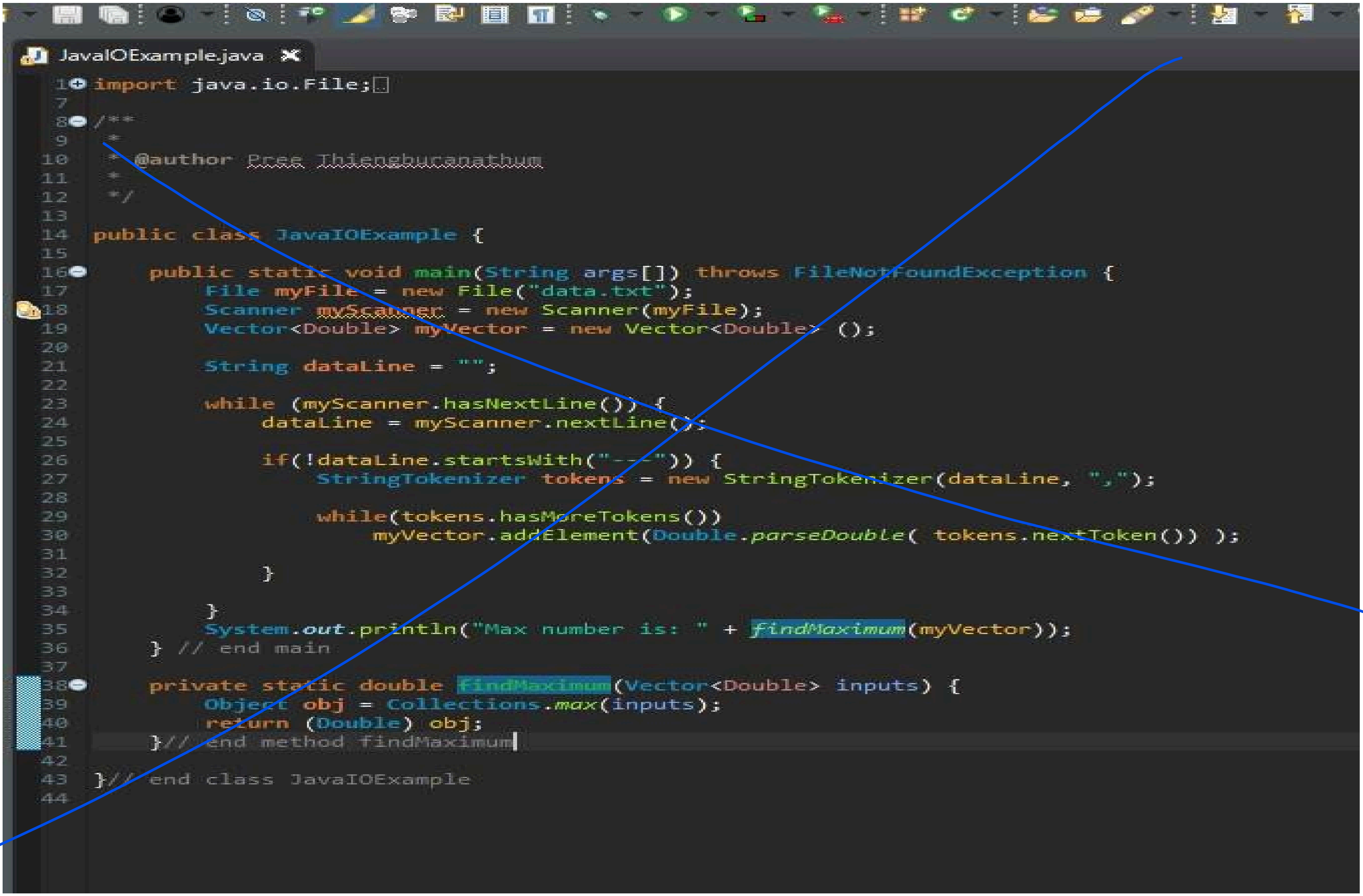
- In general, given n numbers to sort, it performs n^2 comparisons
- The same as selection sort and bubble sort

Class exercise:

Write a java application program that uses Vector class and can output the result as shown in the right.

Output:

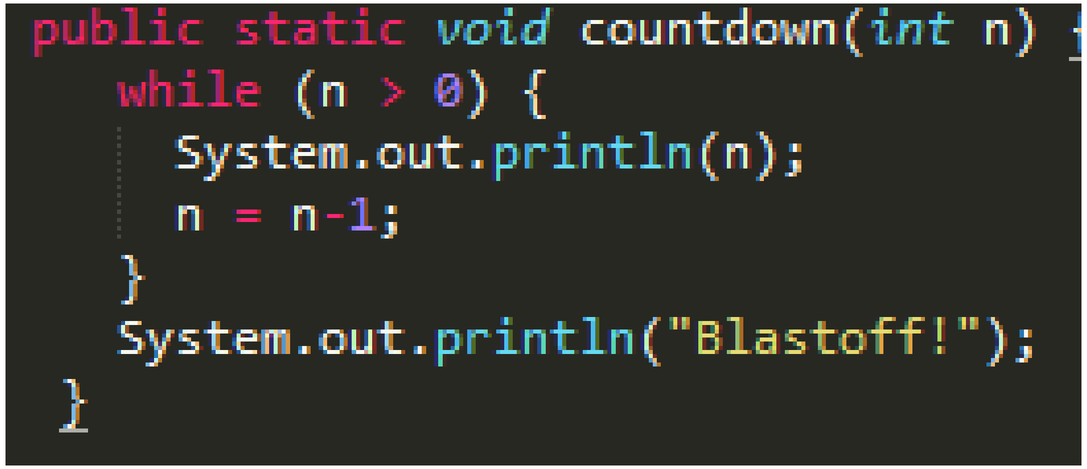
```
Enter lines of input, use quit to end the program.
apple
orange
banana
grape
lemon
quit
Number of lines: 5
First line:   apple
Lines in reverse order:
lemon
grape
banana
orange
apple
```



```java
import java.io.File;

/**
 *
 * @author Pree Thiengburanathum
 *
 */

public class JavaIOExample {

    public static void main(String args[]) throws FileNotFoundException {
        File myFile = new File("data.txt");
        Scanner myScanner = new Scanner(myFile);
        Vector<Double> myVector = new Vector<Double> ();

        String dataLine = "";

        while (myScanner.hasNextLine()) {
            dataLine = myScanner.nextLine();

            if(!dataLine.startsWith("---")) {
                StringTokenizer tokens = new StringTokenizer(dataLine, ",");

                while(tokens.hasMoreTokens())
                    myVector.addElement(Double.parseDouble( tokens.nextToken()) );

            }

        }
        System.out.println("Max number is: " + findMaximum(myVector));
    } // end main

    private static double findMaximum(Vector<Double> inputs) {
        Object obj = Collections.max(inputs);
        return (Double) obj;
    }// end method findMaximum

}// end class JavaIOExample
```

Iterative

- Iterative approach is a repetition process until the condition fails
- They are often simple referred as loops
- Loops are used such as ***for*** loop, ***while*** loop, ***do*** loop.
- By this time you should have master this programming approach.



```java
public static void countdown(int n) {
    while (n > 0) {
        System.out.println(n);
        n = n-1;
    }
    System.out.println("Blastoff!");
}
```

Agenda

- Recursion

Recursion

- Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.
- A function calls repeatedly

- **Advantages:**
  - Performs better in solving problems based on tree structures.
  - Reduce the time to write and debug code, makes code smaller

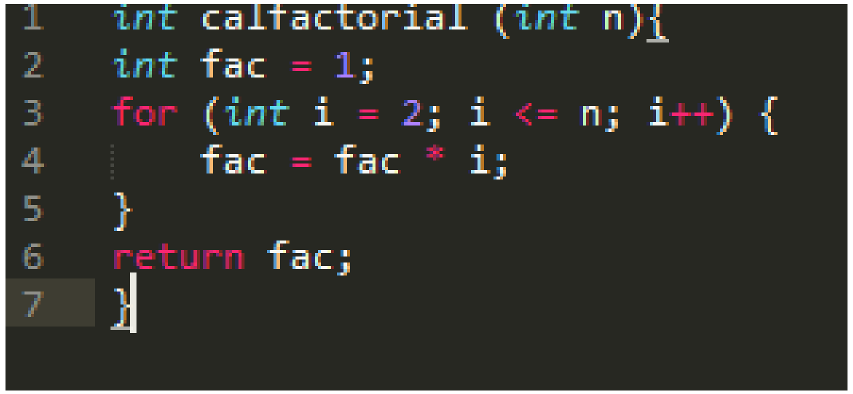
- **Disadvantages:**
  - Use more memory than iterative method.
  - Slower than iterative due to the overhead of maintaining the stack.

Recursion

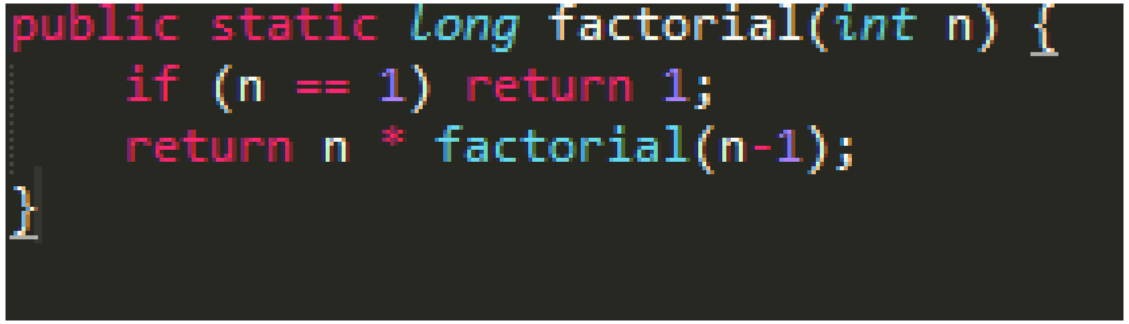
- Recursion is basically divided into two cases:
- *Base case:*
  - It stops the function to call itself when specific condition matches. Whereas there can be more than one base cases.
- *General case:*
  - It call function from within that function.

Example of recursion: Factorial

$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$



```
int calfactorial (int n){
int fac = 1;
for (int i = 2; i <= n; i++) {
    fac = fac * i;
}
return fac;
}
```



```
public static long factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

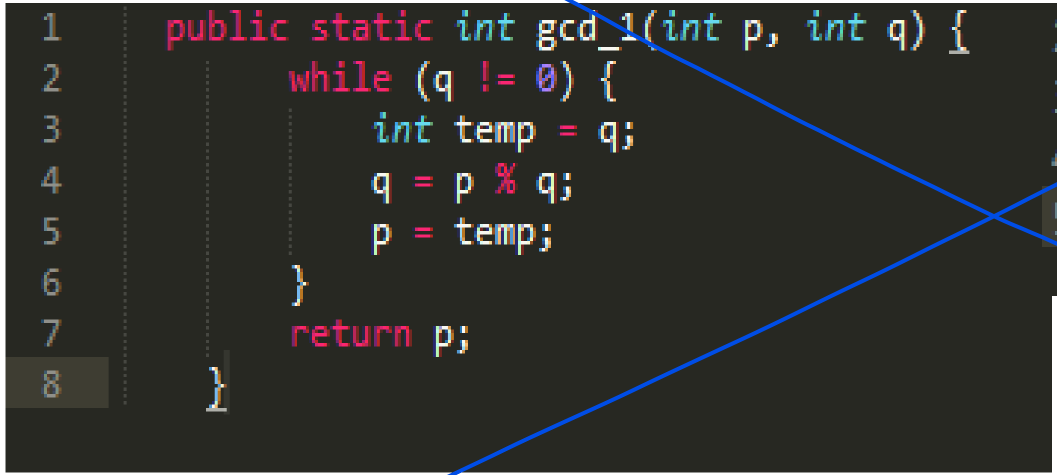
Example of recursion: Euclid’s algorithm

- ~~The *greatest common divisor* (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the *gcd*(102, 68) = 34.~~

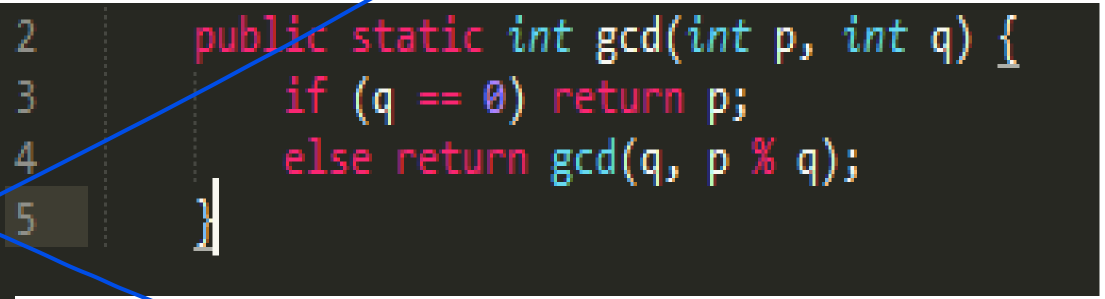
- ~~*gcd*(32, 5)?~~
- ~~32 = 5 x 6 + 2~~
- ~~5 = 2 x 2 + 1~~
- ~~2 = 1 x 2 + 0~~

- ~~*gcd* (3, 7)~~
- ~~7 = 3x2+1~~
- ~~1 = 1x1+0~~

Euclid’s algorithm: iterative vs recurisve



```
public static int gcd_1(int p, int q) {
        while (q != 0) {
            int temp = q;
            q = p % q;
            p = temp;
        }
        return p;
    }
```

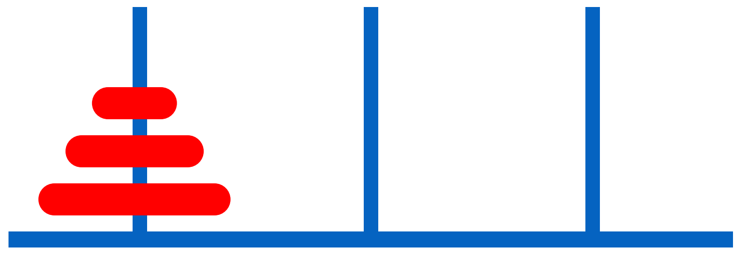


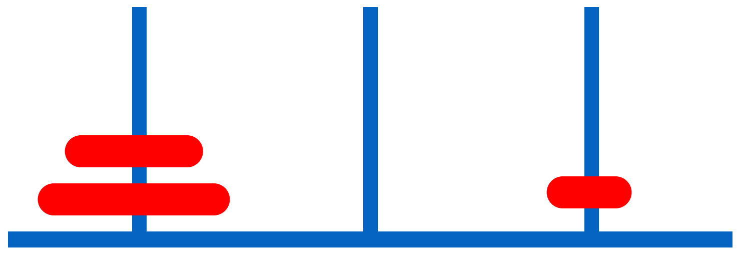
```
public static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}
```

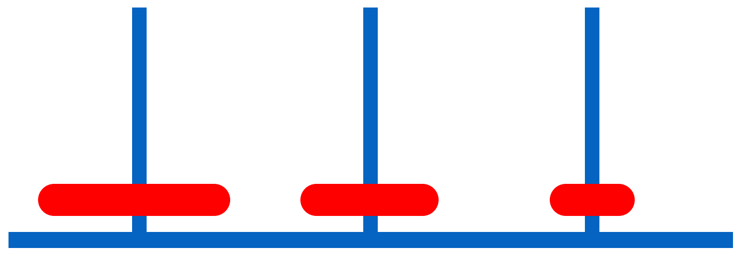
Towers of Hanoi

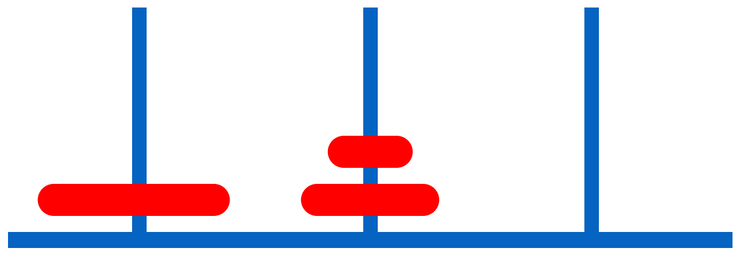
- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on the pegs
- The disks are of varying size, initially placed on one pole with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one pole to another under the following rules:
  - We can move only one disk at a time
  - We cannot move a larger disk on top of a smaller one

Towers of Hanoi

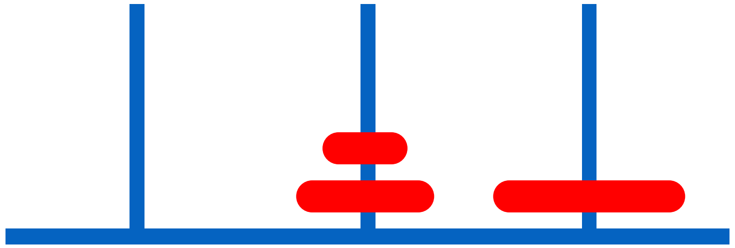

Original Configuration

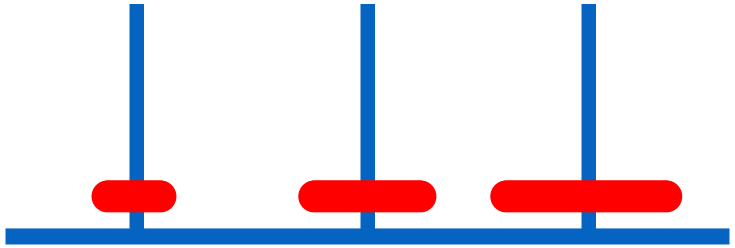

Move 1

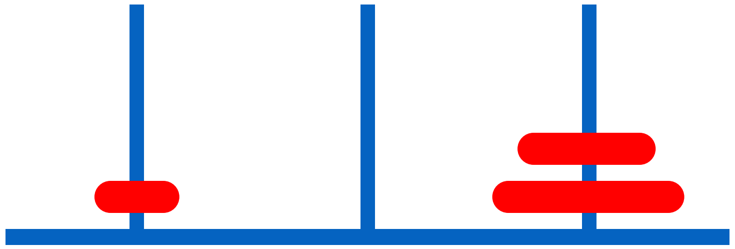

Move 2

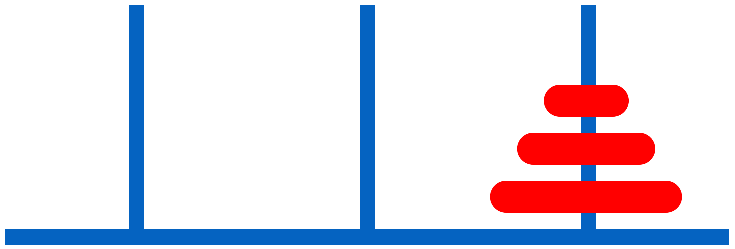

Move 3

Towers of Hanoi


Move 4


Move 5


Move 6


Move 7 (done)

Class exercise : recursion

- Printing problem
  - Write a simple Java program that print number from 20 to 1 for both iterative and recursive version.

Output

```
20
19
18
..
..
1
```