Computer Systems Design and Applications

CS209A

Lecture 2

Adam Ghandar

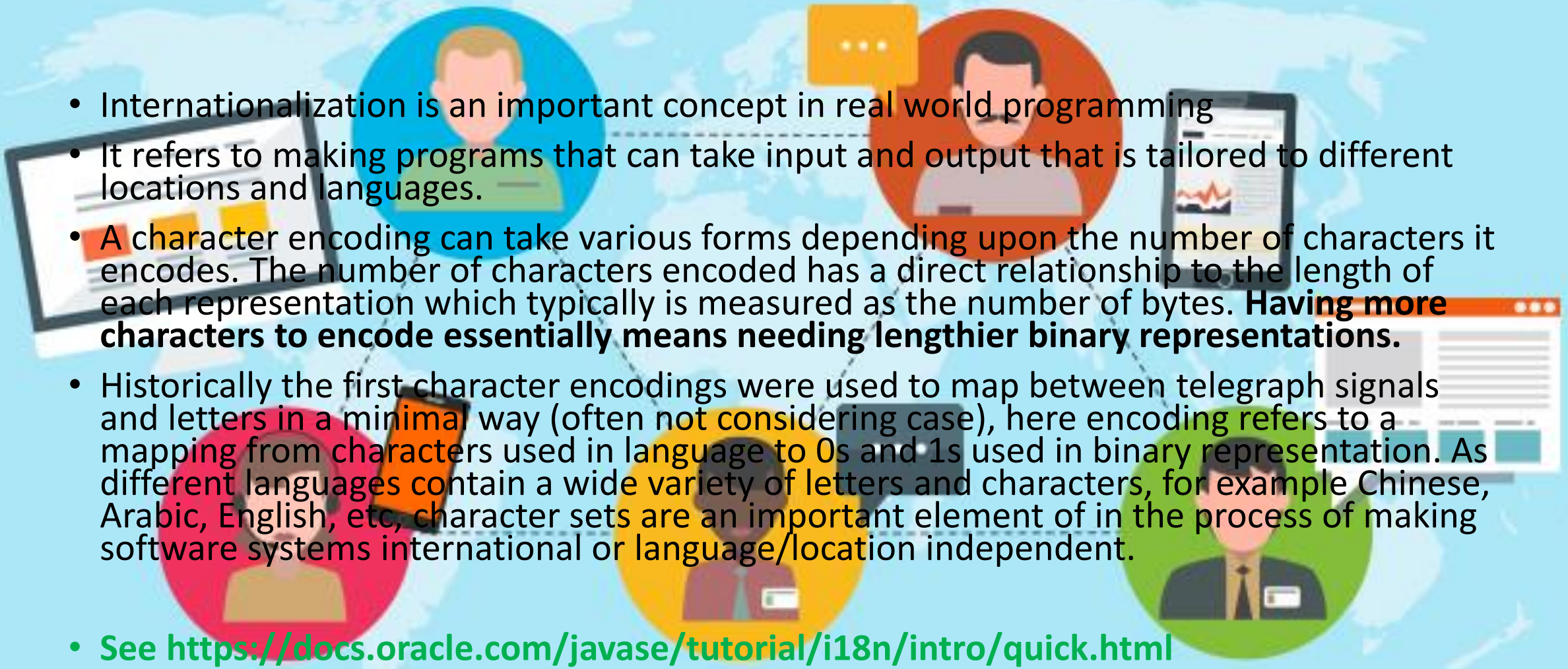
Overview

- Internationalization
- OOP
- Exceptions
- Recursion

Reading

- Continue to read chapter 1 on programming concepts and understand the key points
- Watch the online videos of this chapter from the textbook website (optional but recommended)

Internationalization



- Internationalization is an important concept in real world programming
- It refers to making programs that can take input and output that is tailored to different locations and languages.
- A character encoding can take various forms depending upon the number of characters it encodes. The number of characters encoded has a direct relationship to the length of each representation which typically is measured as the number of bytes. **Having more characters to encode essentially means needing lengthier binary representations.**
- Historically the first character encodings were used to map between telegraph signals and letters in a minimal way (often not considering case), here encoding refers to a mapping from characters used in language to 0s and 1s used in binary representation. As different languages contain a wide variety of letters and characters, for example Chinese, Arabic, English, etc, character sets are an important element of in the process of making software systems international or language/location independent.

- **See https://docs.oracle.com/javase/tutorial/i18n/intro/quick.html**

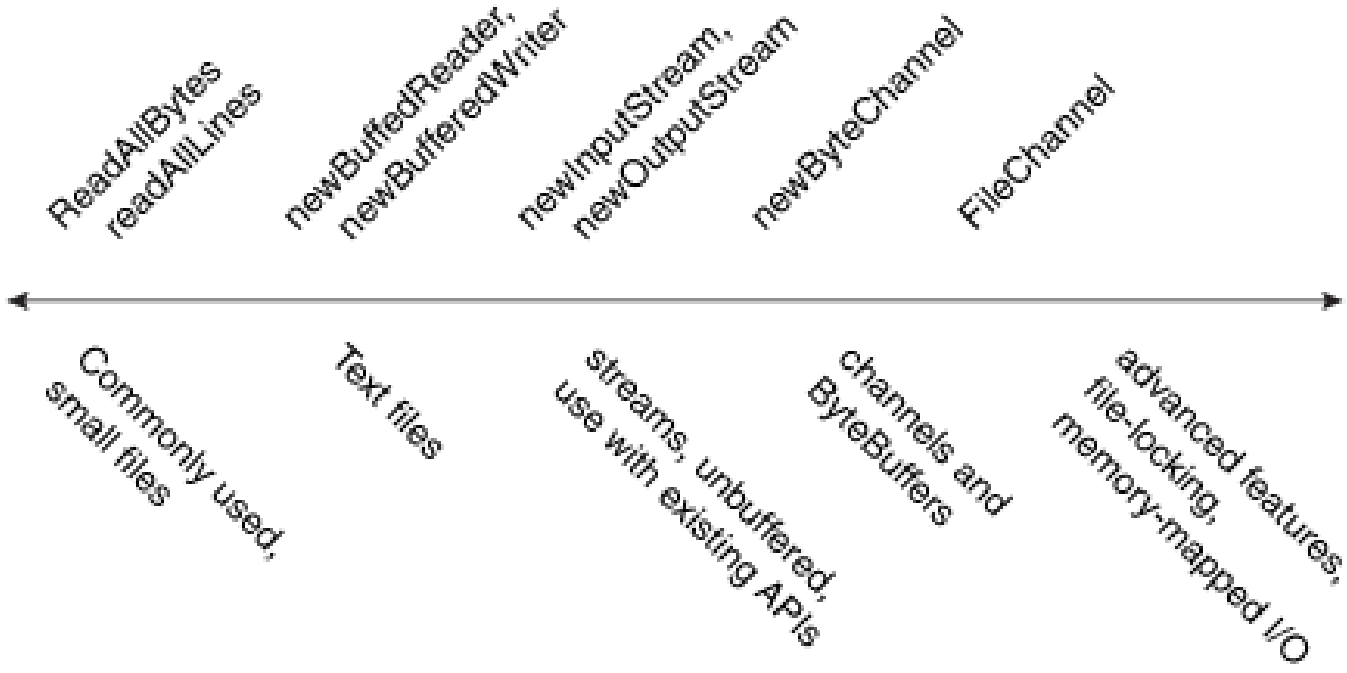
File IO

- In first year you used files to read and save data on a disk

- You can confirm the basics of file IO here (java documentation): https://docs.oracle.com/javase/tutorial/essential/io/file.html

File IO

- There are many file I/O methods to choose from. Here are the file I/O methods arranged by complexity (left is simpler)
- In the lab we emphasize the use of exception handling to facilitate reading and writing from files in Java



Base64 encoding

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

Character Encoding

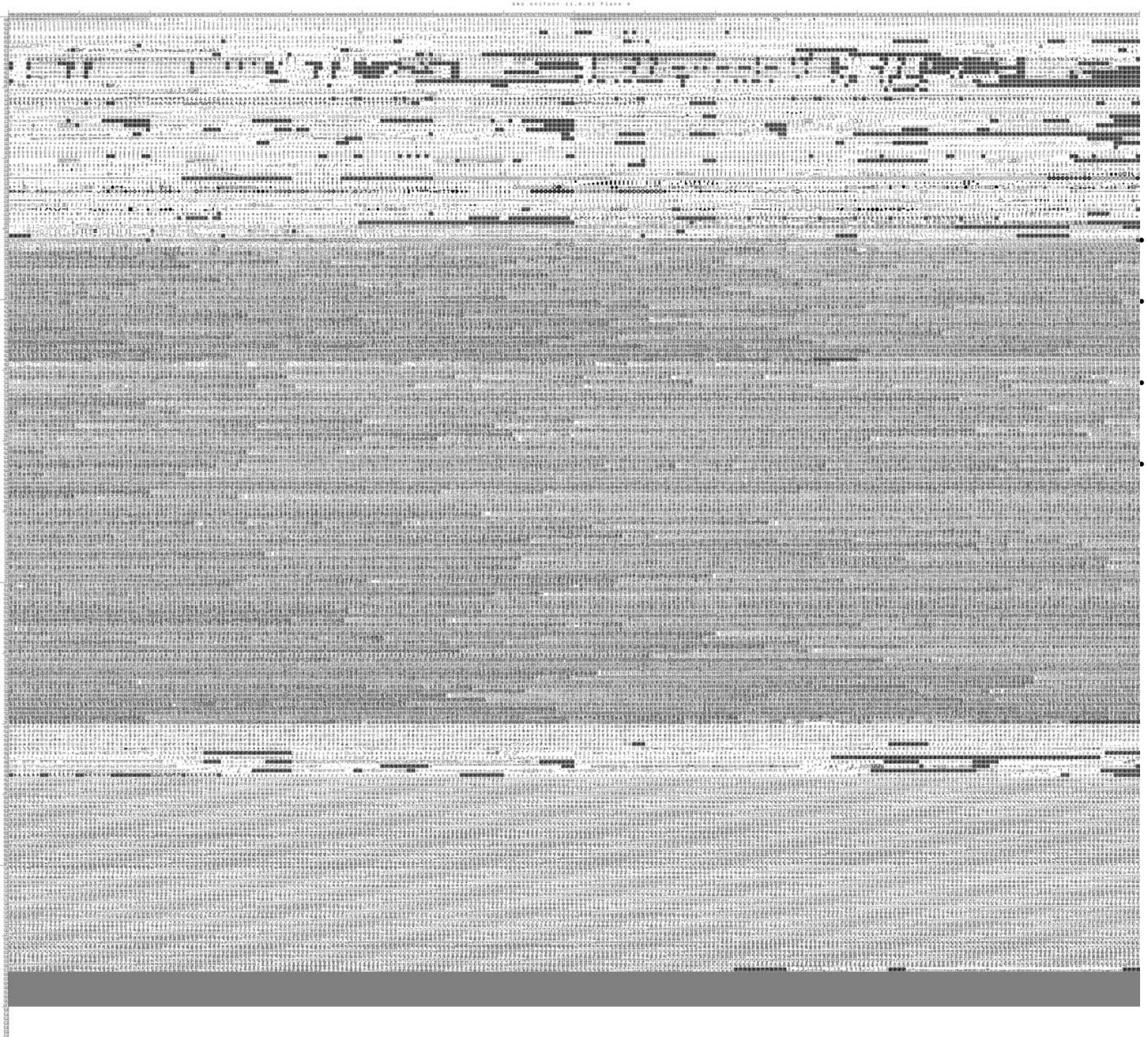
- The International Morse Code encodes the 26 English letters A through Z, some non-English letters, the Arabic numerals and a small set of punctuation and procedural signals (prosigns). There is no distinction between upper and lower case letters.

International Morse Code

- 1 dash = 3 dots.
- The space between parts of the same letter = 1 dot.
- The space between letters = 3 dots.
- The space between words = 7 dots.

A	·−	V	···−
B	−···	W	·−−
C	−·−·	X	−··−
D	−··	Y	−·−−
E	·	Z	−−··
F	··−·	.	·−·−·−
G	−−·	,	−−··−−
H	····	?	··−−··
I	··	/	−··−·
J	·−−−	@	·−−·−·
K	−·−	1	·−−−−
L	·−··	2	··−−−
M	−−	3	···−−
N	−·	4	····−
O	−−−	5	·····
P	·−−·	6	−····
Q	−−·−	7	−−···
R	·−·	8	−−−··
S	···	9	−−−−·
T	−	0	−−−−−
U	··−		

Chinese telegraph code



Unicode

- Different encoding schemes developed in isolation and practiced in local geographies started to become challenging.
- It is not difficult to understand that while encoding is important, decoding is equally vital to make sense of the representations. **This is only possible in practice if a consistent or compatible encoding scheme is used widely.**
- This challenge gave rise to **a singular encoding standard called Unicode which has the capacity for every possible character in the world**. This includes the characters which are in use and even those which are defunct!
- **Therefore, how these code points are encoded into bits is left to specific encoding schemes within Unicode.**

Unicode encodings

- UTF-32 is **an encoding scheme for Unicode that employs four bytes to represent every code point** defined by Unicode. Obviously, it is space inefficient to use four bytes for every character.
  - For example: "T” in "UTF-32” is "00000000 00000000 00000000 01010100” (the first 3 bytes are unnecessary)
- Variable length encoding: UTF-8 is **another encoding scheme for Unicode which employs a variable length of bytes to encode**. While it uses a single byte to encode characters generally, it can use a higher number of bytes if needed, thus saving space.
  - For example: "T” in "UTF-8” is "01010100”
  - But "語” in "UTF-8” is "11101000 10101010 10011110"

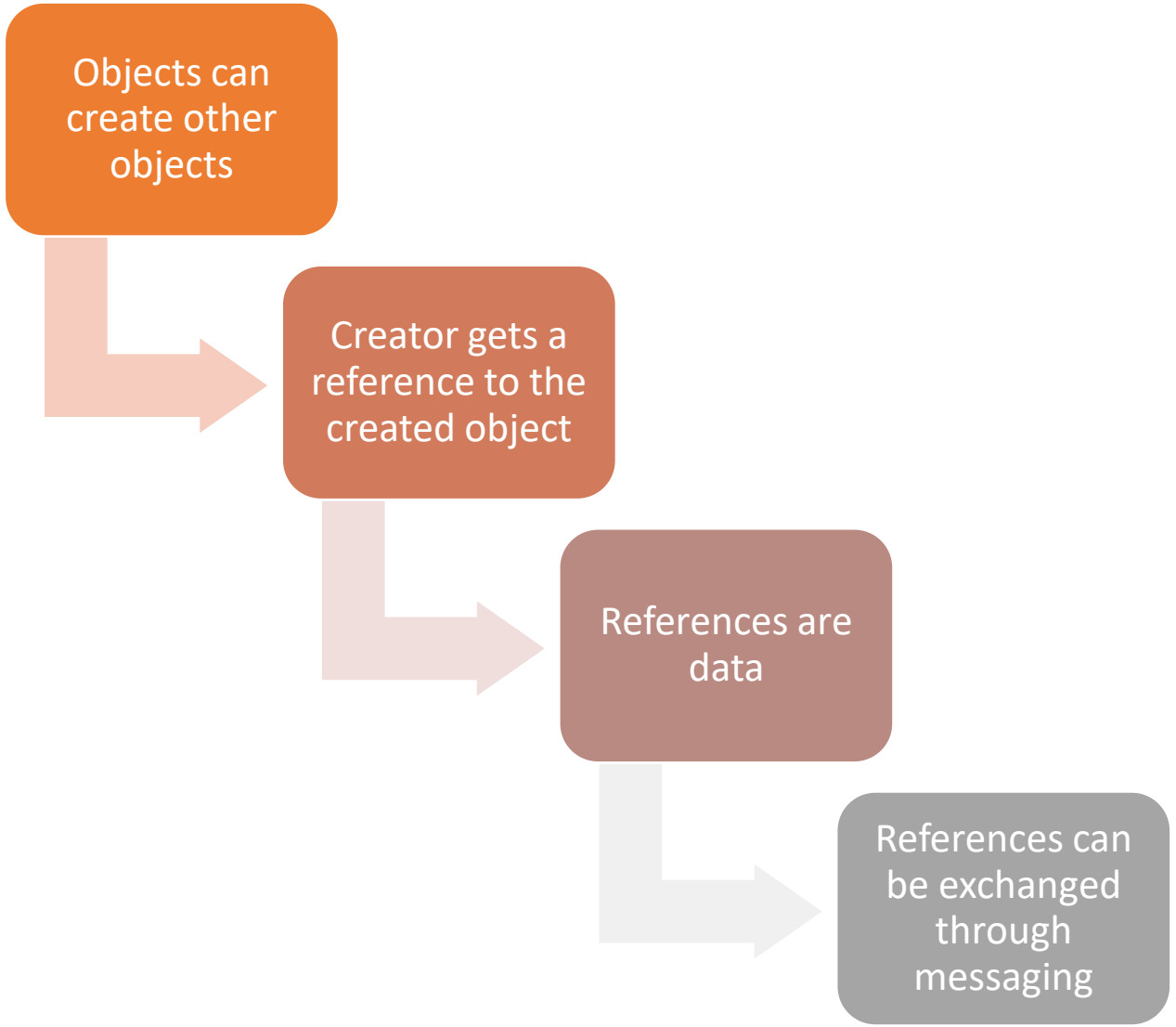
Summary

- FileIO, Internationalization, and Charset encoding are important practical concepts last week we looked at some examples in the lab.

OOP

Object References

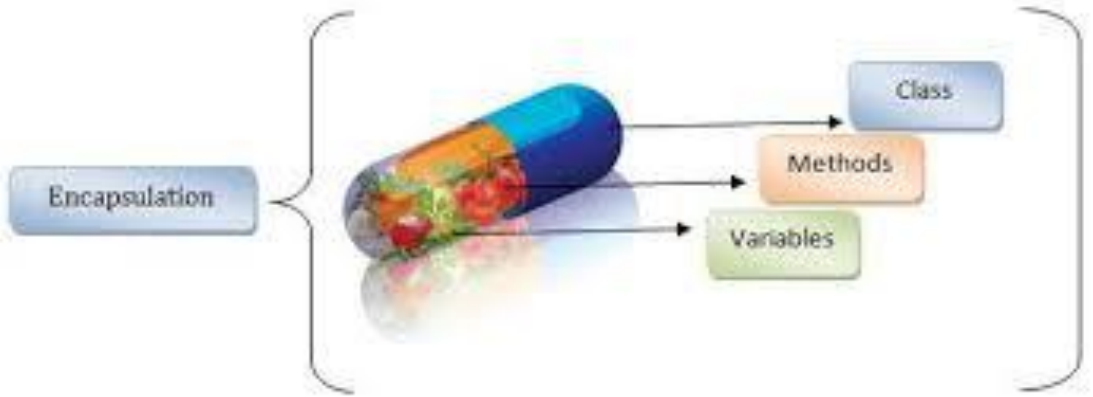
- You need the identity (reference) of an object to exchange a message (call a method) with it. An object can create another object by invoking its "constructor" and the constructor returns the reference. But the creator may not be the only object that expects services from the newly created object. In many cases, it will pass around the reference (as a method parameter) so that the other objects it requests a service from can use services from this new object.
- It's not recommended to let every object create objects, and it's considered to be a better practice to have objects that act as "object factories" and then pass along references. This is a technique known as "dependency injection".



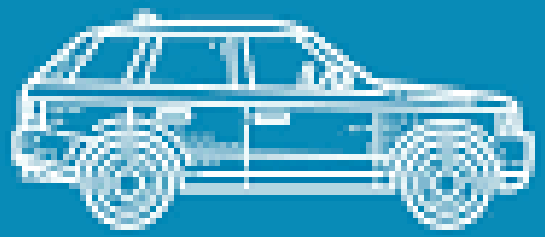
You mustn't forget some of the key principles of object-oriented programming: encapsulation, which means that everything is private except the public methods that define the interface of the object with the outside world, and responsibility.

KEY PRINCIPLE: Encapsulation

Responsibility – the object has all the means to provide the service



Encapsulation



Car

model

speed

engine

speedLimit

drive()

stop()

setSpeed(number)

Designing OO Programs

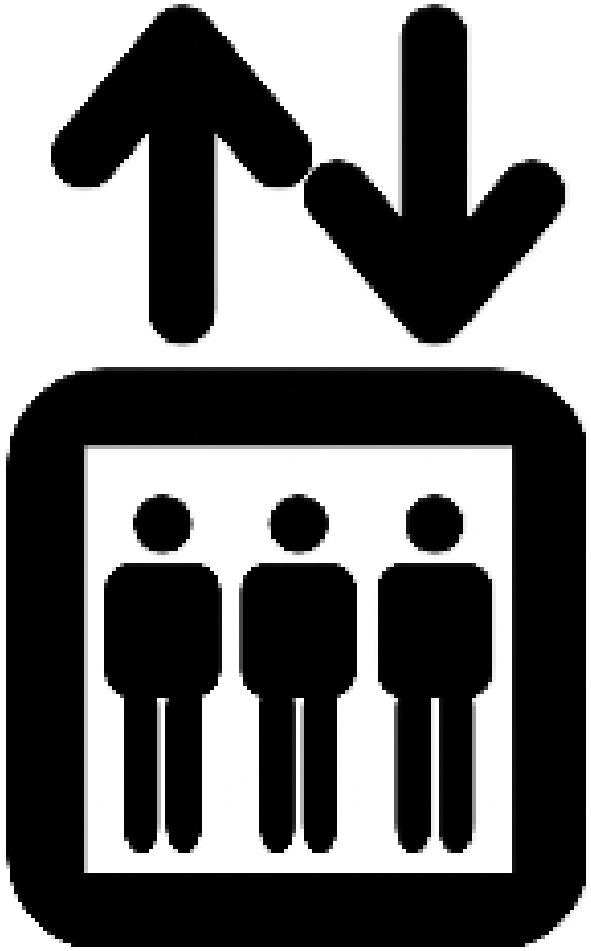
One of the problems in object oriented programming is that some of recommended strategies are at odds with each other.

Small objects doing small tasks are better for consistency (which means that objects are focused on one task), reuse, and testing (you can write a mock object that returns hard-coded values while someone is debugging the real thing); but they increase communications and coupling.

- Better to have several small objects than a big one (testing) - consistency
- Better to pass references than leave *any* object create new objects
- Better to minimize communications (weak coupling)

Challenges

- In theory, Object Oriented programming is easy. There is a saying that "the gap between theory and practice is wider in practice than it is in theory". One (but perhaps no the greatest) difficulty is to correctly identify objects. In spite of their name, "objects" don't always correspond to real-life objects; sometimes they implement a function. Let's illustrate it with an example.



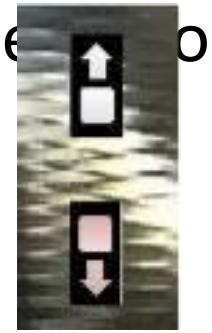
Lift Example

Problem Description

If you work for a lift manufacturer, you might have to code software for the operation of lifts. Optimizing the movements of lifts so as to minimize waiting times is not a small task. Knowing how many lifts are required in an office building depending on the number of people working in this building can also be challenging. One complicating factor is the need to model peak usage as people tend to use the lifts at the same time. Modelling and simulation can be used to do this, which takes us back to the roots of Object-Oriented programming.

Implementation

We may think of creating a lift class. First of all, it must know which floors it serves. Then, it has a state: it may be stopped (for maintenance sometimes) or moving, up or down, and won't change direction before it has reached an "extreme stop". Messages are of two kinds: an "up" or "down" call from the outside (floor is known) and a "go to floor" call when people press a button inside.



Lift Class

- Floors served
- Direction
- Moving/Stopped
- Stop request

This model works when there is a single lift, but not when you have an "elevator lobby" served by several lifts. When you call a lift, you care very little about which lift will stop at your floor as long as it goes into the right direction.

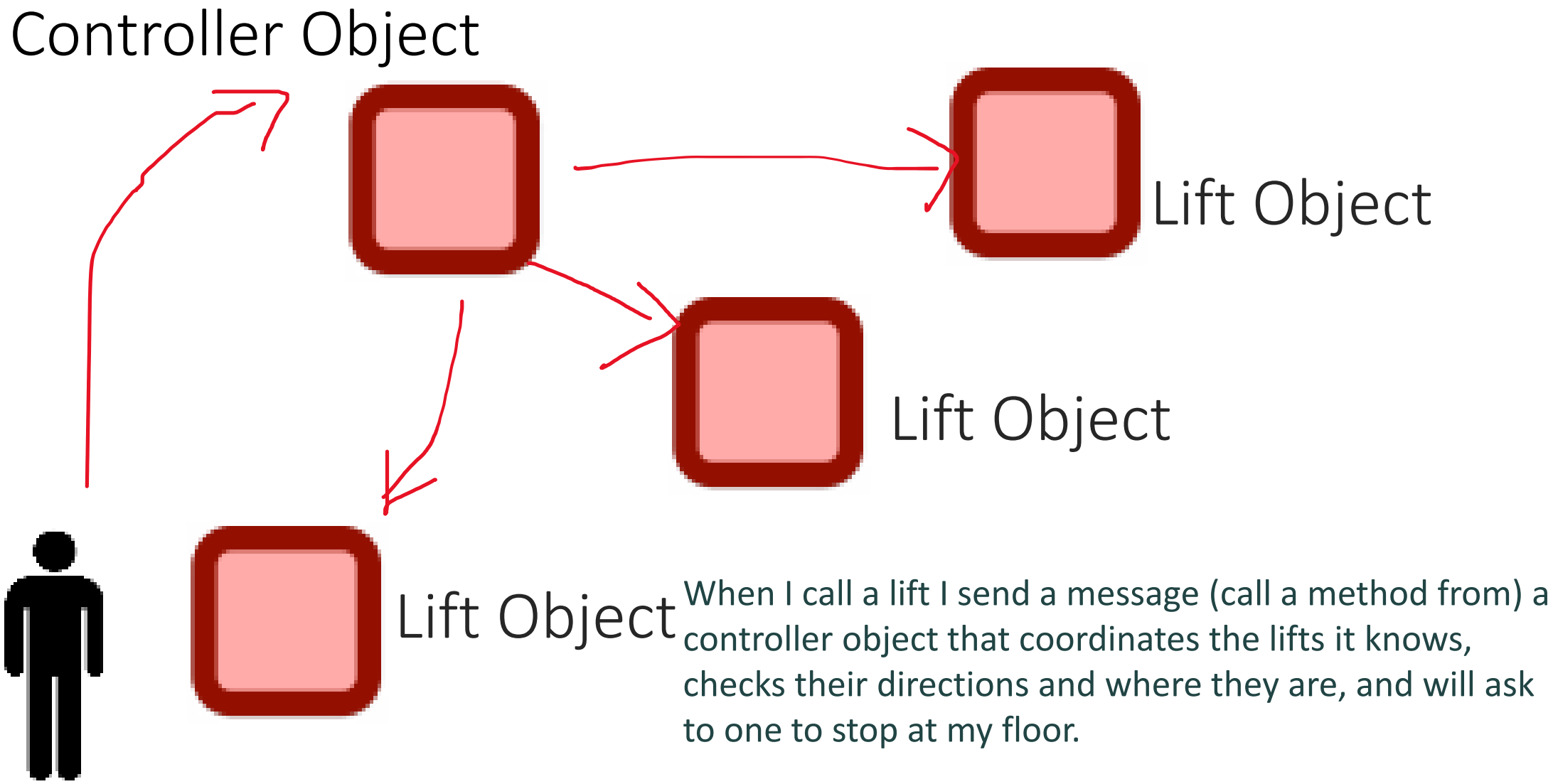
Controller Class

- Lift Requests

In that case we need a more abstract class, which I call "Controller class", for coordinating several lifts and taking call requests. Buttons inside the lift are still messages to a lift object that represents the lift you are in.

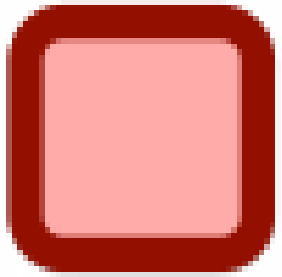
Lift Class

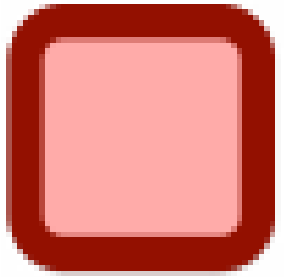
- Floors served
- Direction
- Moving/Stopped
- Stop request



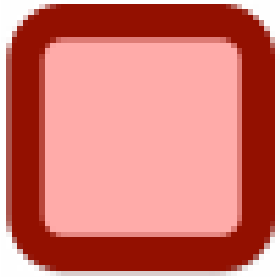
When I call a lift I send a message (call a method from) a controller object that coordinates the lifts it knows, checks their directions and where they are, and will ask to one to stop at my floor.

Controller Object

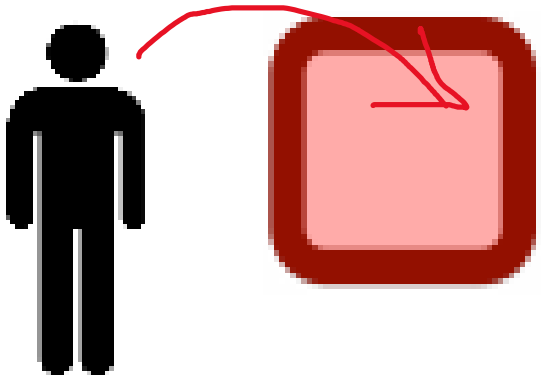




Lift Object



Lift Object



Lift Object

When a lift stops at my floor, I'll step into it and tell to that particular lift where I want to go (which the system doesn't know so far) by pressing a button inside the lift. And here is my object application.

Creating an Object Oriented Application

- Identify necessary objects
- Define their role (what they do)
- Define the data they need
- Set up communications

- **Maximize consistency and minimize coupling**

the hard part

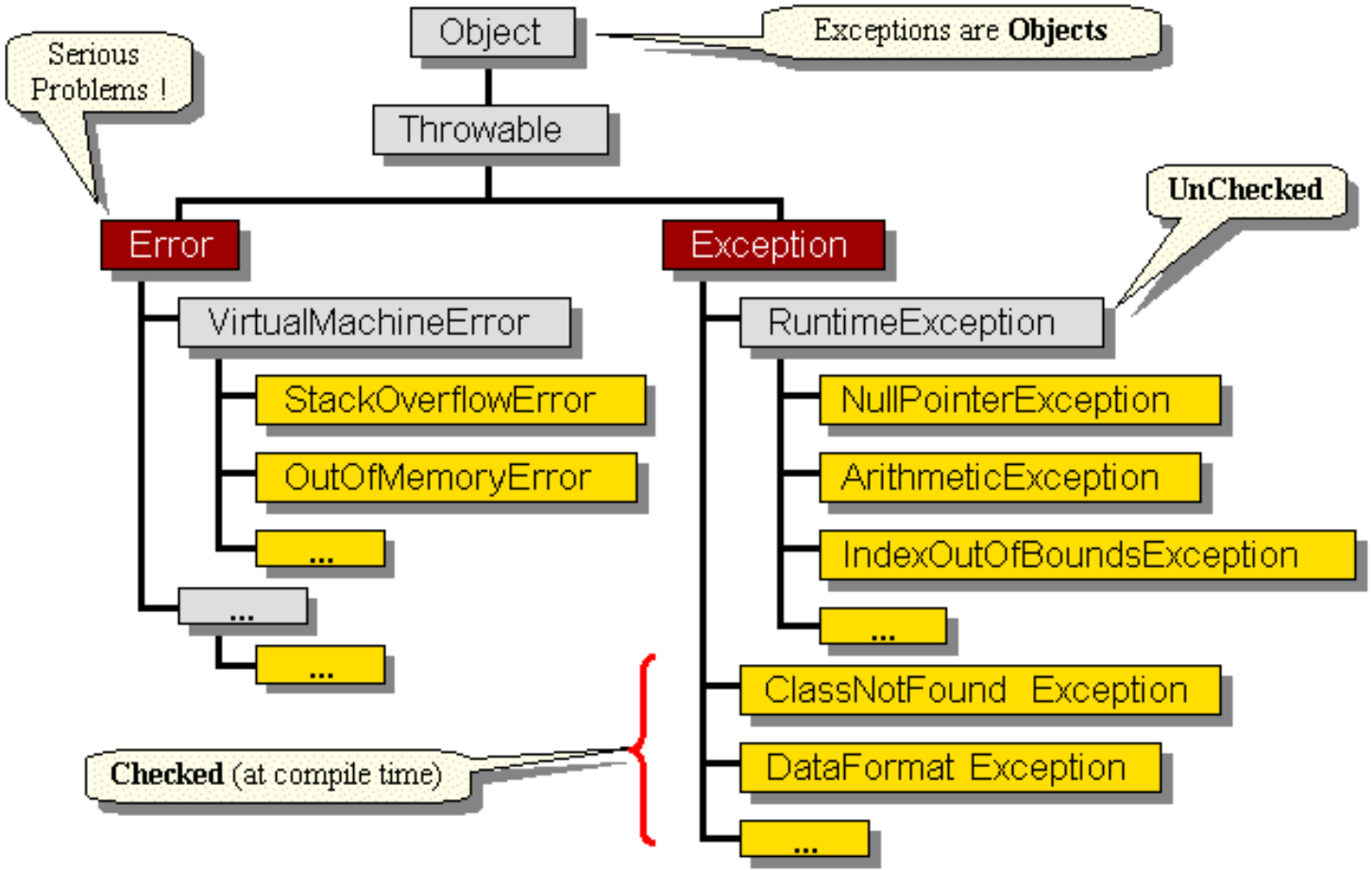
- Trade-off between one object that does everything and objects that send too many messages

Exceptions VS Errors

- Exceptions are abnormal conditions that occur in your program or java telling you something has gone wrong. When an exception occurs java forces us to handle it (need to know when and what to do) or declare we want someone else to handle it.
- Exception handling provides a way to handle and recover from errors or a way to quit the program in a graceful way.
- Errors represent serious represent serious unrecoverable problems for example VirtualMachineError.OutOfMemoryError

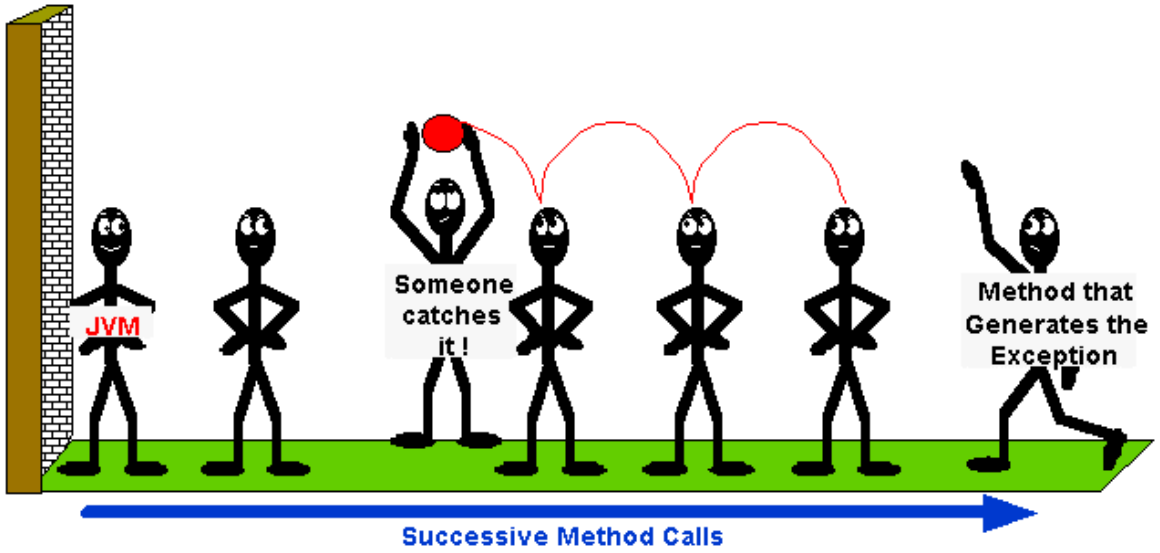
Types of exception and error cases

- Checked exceptions are checked by javac at compile time (e.g. class not found)
- Unchecked exceptions are not checked at compile time (Null pointer, index out of bounds, arithmetic)
- Errors are usually kept for major problems that cannot be recovered from (e.g. stack over flow, out of memory)



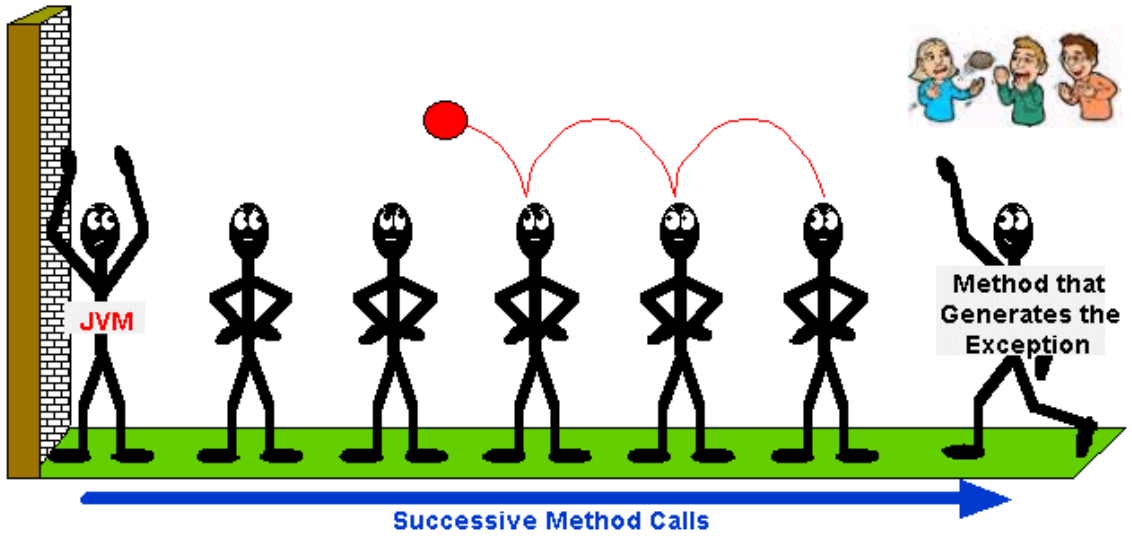
Exceptions VS Errors

Throwing Exceptions



- …
- public void getInformation() throws MyException {
- …
- }
- private void doStuff(){
- try {
- getInformation();
- catch (MyException ex){
- //handle here
- }

Throwing Exceptions



- …
- public void getInformation() throws MyException {
- …

So you can either catch exceptions, or pass them back to the caller. If it's a checked exception and you don't catch it, you must declare that the method can throw the exception.

But… Something else can happen

Compile-time

Link-time

Run-time

Logic

- What if you have logic errors in your program?
- Java cannot see these types of errors

Program terminates with an incorrect result

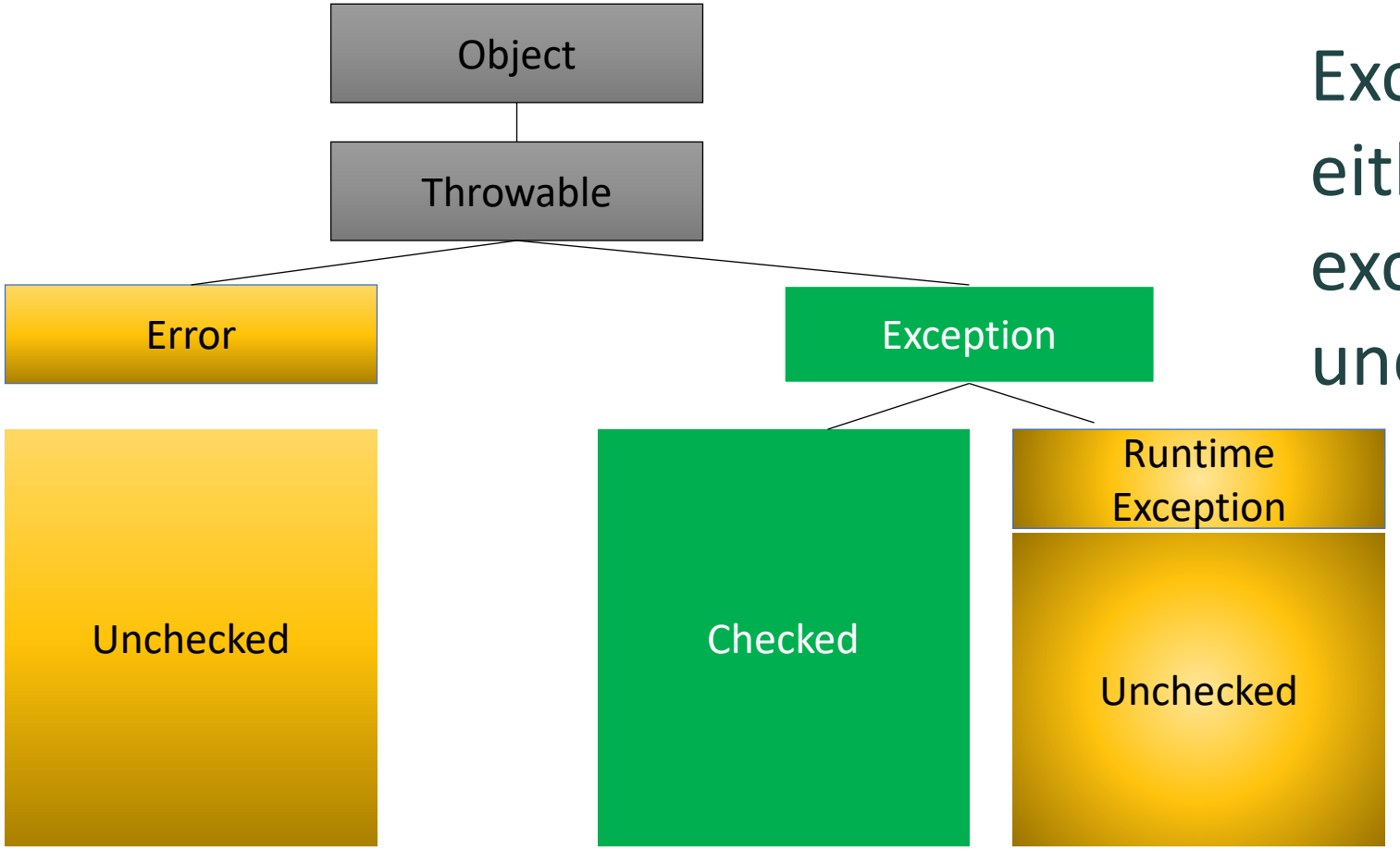
Program crashes because of divide by zero error in a calculation
Or… Null object references
Or… Access an array with an out of bounds index

Logic Errors

- These types of errors are not really expected happen
- They are a major item that are in "**Unchecked Exceptions**"
  - Errors are also unchecked
  - Runtime exceptions

No requirement for catching or declaring by the compiler

Checked and Unchecked Exceptions



Exceptions can be either checked exceptions or unchecked exceptions

Unchecked Exceptions

- A method is not forced by compiler to declare the unchecked exceptions thrown by its implementation. They will come into life once buggy code is executed.
- Examples:
  ArithmeticException, ArrayStoreException, ClassCastException

Why is this important?

This is important because you often want to create your own exceptions in an application when the application logic / business rules are violated (for example in a bank application a negative account balance may generate an exception).

➢ *You can create your own exceptions*

```
…
} catch (MyException e) {
    …                                …
}
```

handling

Creating New Exception Types

- You don’t create your own exceptions from scratch but extend a Java Throwable so you have to decide what class you will extend.

```
class MyException extends Exception {

 …

}
```

- **Choices**: should you extend Exception? Then it will be a checked exception and javac will ensure it is used according to the rules…

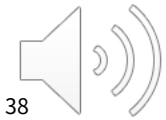
Creating New Exception Types

Unchecked Exceptions are subclasses of RuntimeException

```
class MyException extends RunException {
 …
}
```

Not Checked

- **Choices**: Or will you extend RuntimeException and then javac will not force users of your exception to follow the requirements of declaring it in the method name and using throw or try/catch

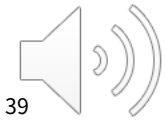


Is an event an exceptional event??

- Depends how often the event/case happens…

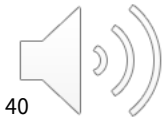
- Unlikely? “once a week?”
- Shouldn’t happen? “once every 5 years?”

Care is necessary: for instance in big transaction system one in a million may be every day



Always check?

In theory, checked is better as javac will ensure that all the users of your method work properly. However in practice it may be different; if you are modifying an existing class already used in dozens of programs, you don't want to add an exception that will require all these programs to be modified (and tested) to take it into account (you can also assume if everything has run without this exception so far, it can go unchecked). Or you may want to provide a "wrapper" method that catches the exception.



Is it reasonable to recover?

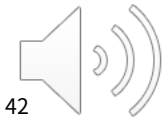
- One convention for deciding between checked an unchecked exceptions is for **checked** exceptions to be used where the case is *predictable but also unpreventable* **and** *reasonable to recover from*

- **Unchecked** exceptions should be used for everything else

Predictable but also unpreventable

Method caller did validate the input parameters however some condition outside their control has caused the procedure to fail.

Example: attempting to load file that has been deleted it between the time you check if it exists and the time the read operation commences. Here by declaring a checked exception, you are telling the caller to anticipate this exceptional case



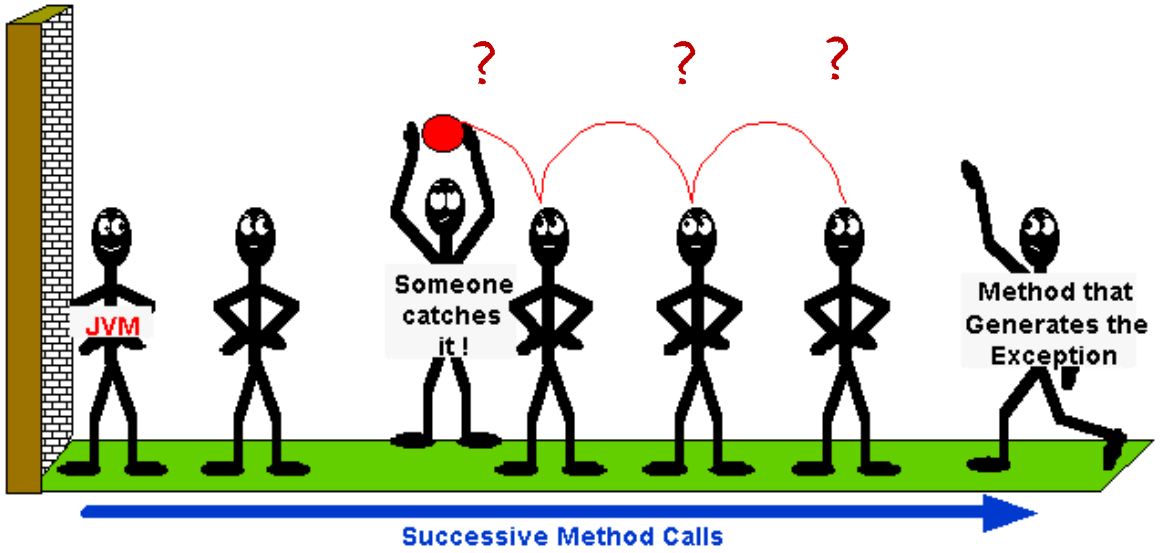
Reasonable to recover from

It is pointless to force callers to anticipate exceptions that they cannot recover from.

If a user attempts to read from a non-existing file, the caller can prompt them for a new filename. On the other hand, if the method fails due to a logic error arising from a programming bug (invalid method arguments or buggy method implementation) there is nothing the application can do to fix the problem in mid-execution.

One thing could do is to log the exception and report to support team (more later).

Who catches it?

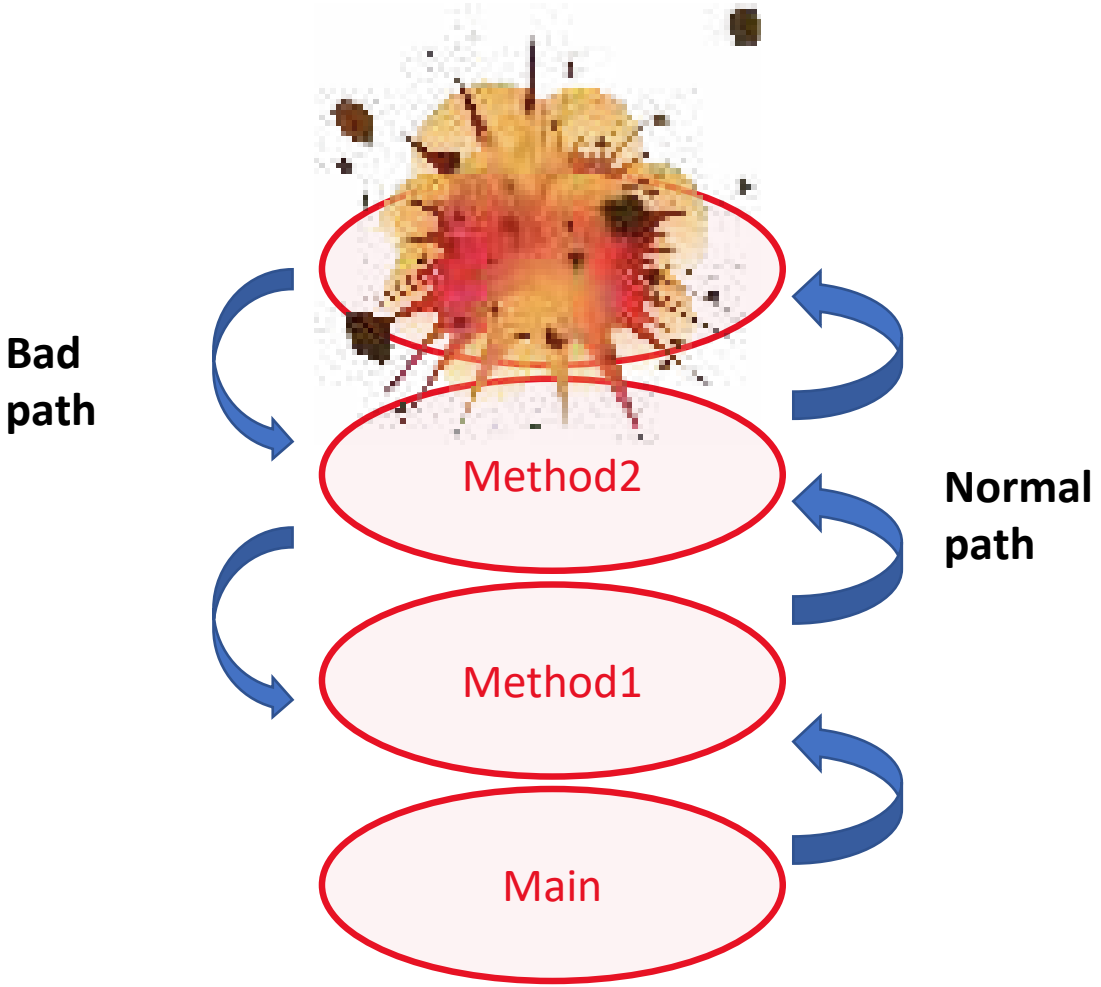


Choose the right abstraction level

Exception wrapper

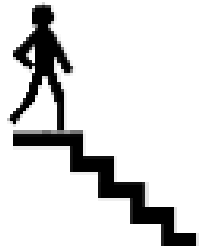
For both checked and unchecked exceptions, **use the right abstraction level**. For example, a code repository with two different implementations (such as a database and a filesystem storage) should avoid exposing implementation-specific details by
throwing SQLException or IOException. Instead, it should wrap the exception in an abstraction that spans all implementations (e.g. RepositoryException).

Exception Processing

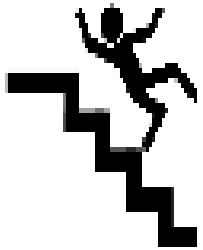


Whether your exception is checked or unchecked, you have to take into account what happens when it's thrown.

Exception Processing



```
try {
…
        WalkDownStairs();
} catch (ExceptionType fallover) {
…
        GrabHandRail();
}
```



- Everything is fine case
- And if you catch the exception you need to thing about what to do when things turn bad as they often do.

Exception Processing

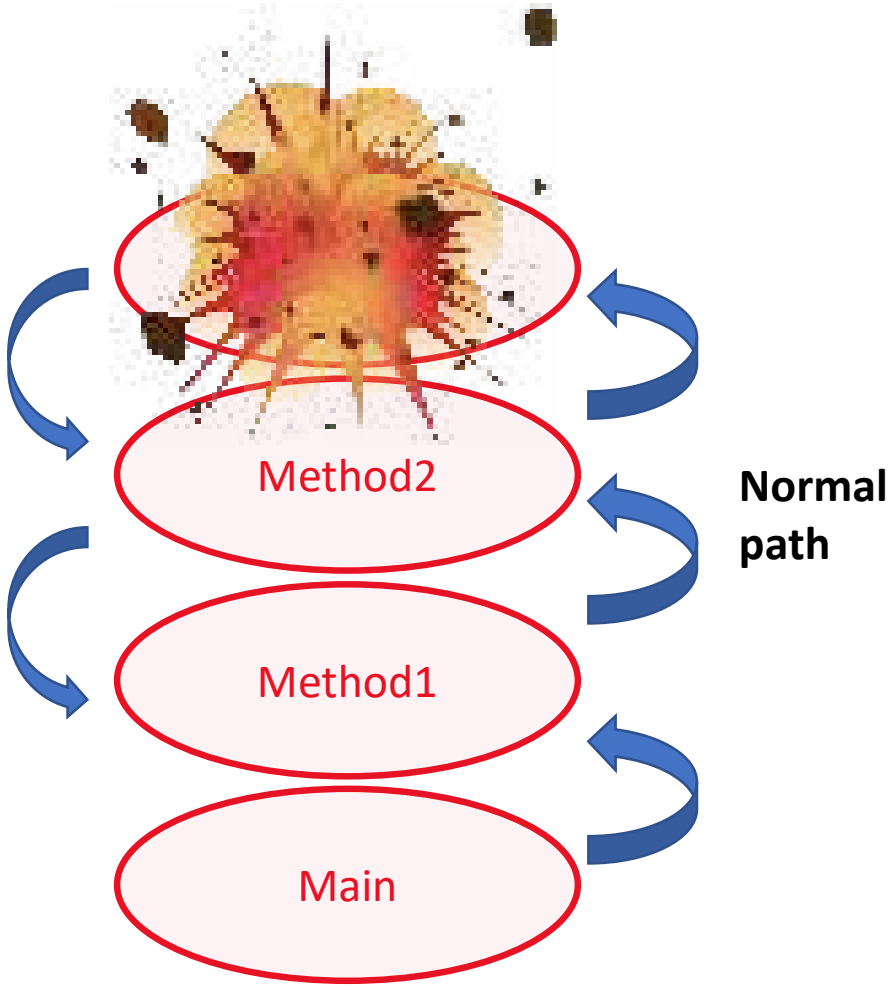
```
try {
        …
} catch (ExpectedException e1) {
        …
} catch (ExpectedException e2) {
        …
} catch (…)
        …
}
```

It can become complex, especially when deciding about how the application should behave next.

Unknown unknowns

- Unexpected and not known or checked
- Also called “Unknown unknowns”

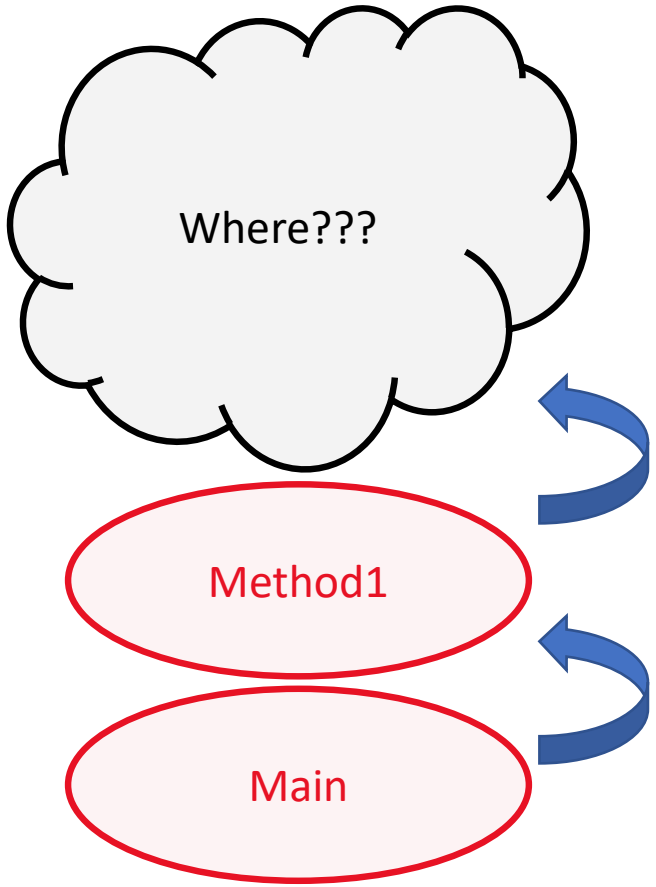
Exception Processing



One possibility is to just ignore exceptions and let them “bubble up”

Here we let method1 deal with the problem that happened in method3

The problem




```
} catch (Exception e) {
            System.err.println("Error: %s", e.getMessage());
...
}
```

- The problem is errors that happen too often have to be fixed
- Here method2 might call many methods apart from method 3 and we have to find where the problem is

Throwable

Throwable()
Throwable(String message)
Throwable(String message,
Throwable cause)
Throwable(Throwable cause)

String getMessage()
Throwable getCause()
Void printStackTrace()

The throwable class has constructors that can help to make it easier to track down errors

NOTE: these constructors are in Throwable and not in the Error or Exception classes

Handling unknown unknowns

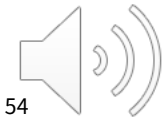
`Option 1:`

Just ignore them – one possibility is when something really unexpected happens just ignore it

Handling unknown unknowns

`Option 2:`

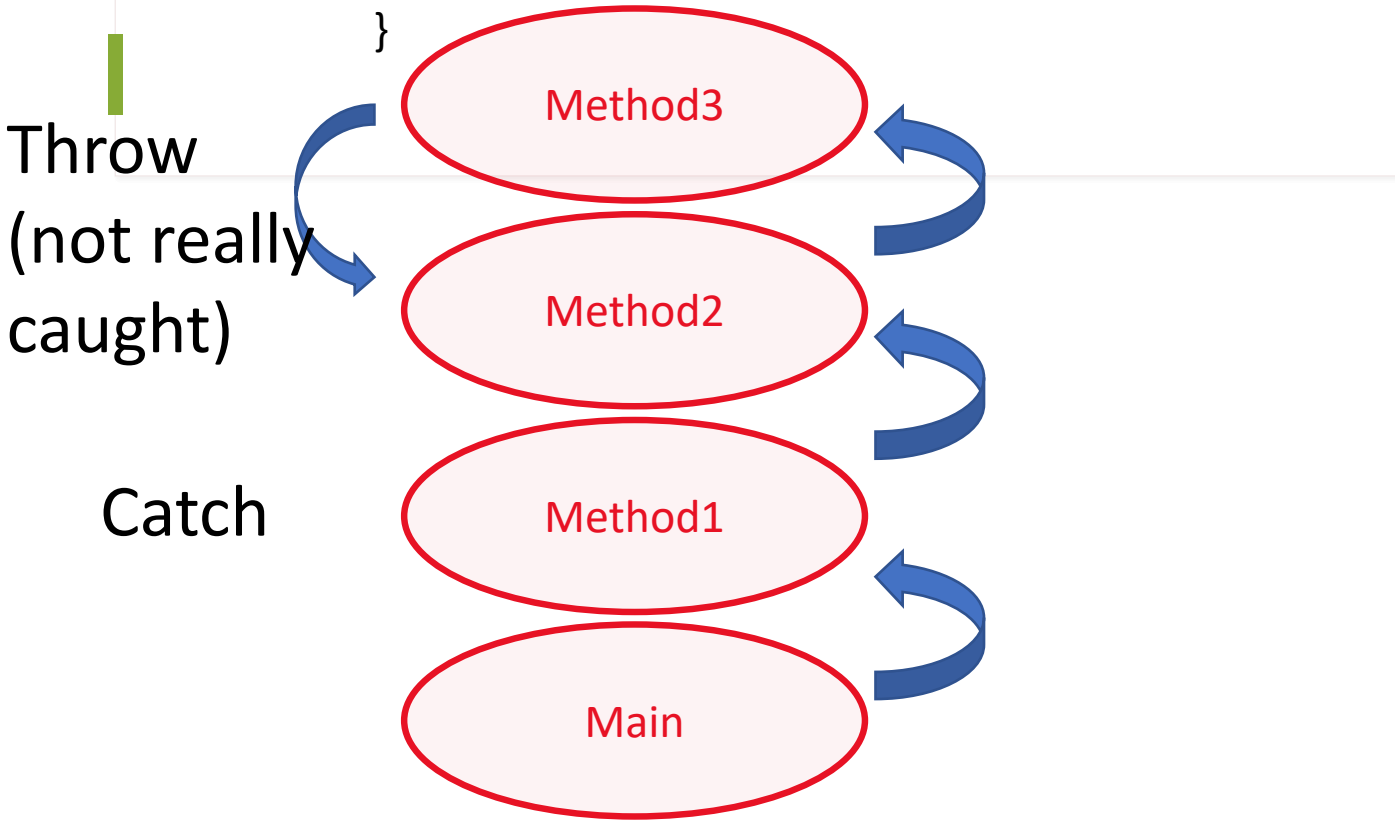
Pass up information about the exception and how it occurred (cause)



Processing exceptions

- A method can catch any exception and then throw it again with a message that says the exception was noticed

```
throws Exception
} catch (Exception e) {
        throw new Exception (“method2”, e)
}
```

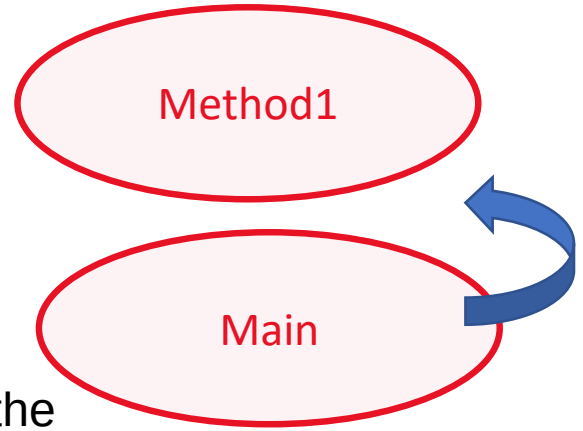


Stack Trace

```
} catch (Exception e) {
StackTraceElement elements[] = e.getStackTrace();
for  int i = 0, n = elements.length; i < n; i++) {
   System.err.println(elements[i].getFileName()
      + ":" + elements[i].getLineNumber()
      + ">> "
      + elements[i].getMethodName() + "()" );
}
                  }
```

Catch

- Processing stack trace information



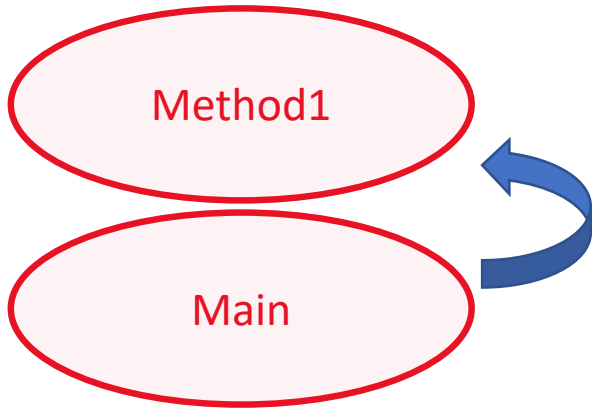
Definition: A *stack trace* provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown. (java docs)

Chained Exceptions

```
} catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        Throwable t = e.getCause();
        while (t != null)
                System.out.println("Cause: " + t.getMessage());
                t = t.getCause();
        }
        …
```

- When we catch the exception we can display the full list of things that wen wrong

Catch



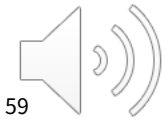
Exception Processing

```
try {
        …
} catch (ExpectedException e1) {
        …
} catch (ExpectedException e2) {
        …
} catch (Throwable t)
        // unknown and unexpected
        …
}
```

You could ignore exceptions by catching them and doing nothing

NEVER
IGNORE
the
unexpected

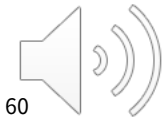
You should only ignore things that you know aren't important – for instance, getting past the end of a string when you are done scanning it ...



Error Messages

- Don’t SCARE end users with excessive technical detail in messages

- $ java Prog
  Enter an integer value: A Exception in thread "main"
  java.util.InputMismatchException
- at java.util.Scanner.throwFor(Scanner.java:864) at
  java.util.Scanner.next(Scanner.java:1485) at
  java.util.Scanner.nextInt(Scanner.java:2117) at
  java.util.Scanner.nextInt(Scanner.java:2076) at Prog.main(Prog.java:8)
- $



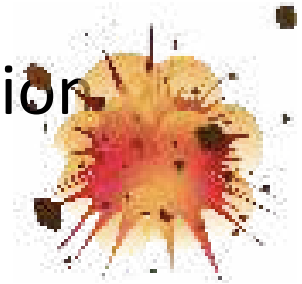
Error messages

- But provide enough detail for the support team

.. and providing enough information either for the user to correct the problem by oneself ("Invalid number" in the previous case) or to the people who in many companies are here to help customers or colleagues. A short error code or message saying what went wrong would help them.

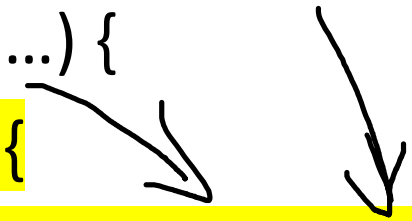
Finally some operations need to always be performed

```
try {
        open a network connection
        send a message
        close the connection
} catch (…) {
}
```



Finally some operations need to always be performed

```
try {
        open a network connection
        send a message
        this is the purpose of finally – it is executed in all cases
} catch (…) {
} finally {
        if connection opened, close it
}
```



Some operations have to be performed in all cases

There is a new alternate syntax in java versions greater than 7 (java >= 7)

"try with resources"

Since Java 7, there is a new syntax known as "try with resources". You can just after try instantiate a new object. If this object has a close() method, and if it implements a special interface, close() will be called automatically at the end of the try {} catch {} block, just as if this method had been called in a "finally" block.

Declarations must implement the (auto)closable interface

```
try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");

        System.out.println(coffeeName + ", " + supplierID + ", " +
                           price + ", " + sales + ", " + total);
    }
} catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
}
```

Call.close() is executed when exiting the try-catch block

```
try {
            Statement stmt = con.createStatement()
   ResultSet rs = stmt.executeQuery(query);


   while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
      int total = rs.getInt("TOTAL");


      System.out.println(coffeeName + ", " + supplierID + ", " +
                  price + ", " + sales + ", " + total);
   }
} catch (SQLException e) {
   JDBCTutorialUtilities.printSQLException(e);
}finally {
try {
stmt.close();
} catch (SQLException e) { // do nothing

   }
}
}
```

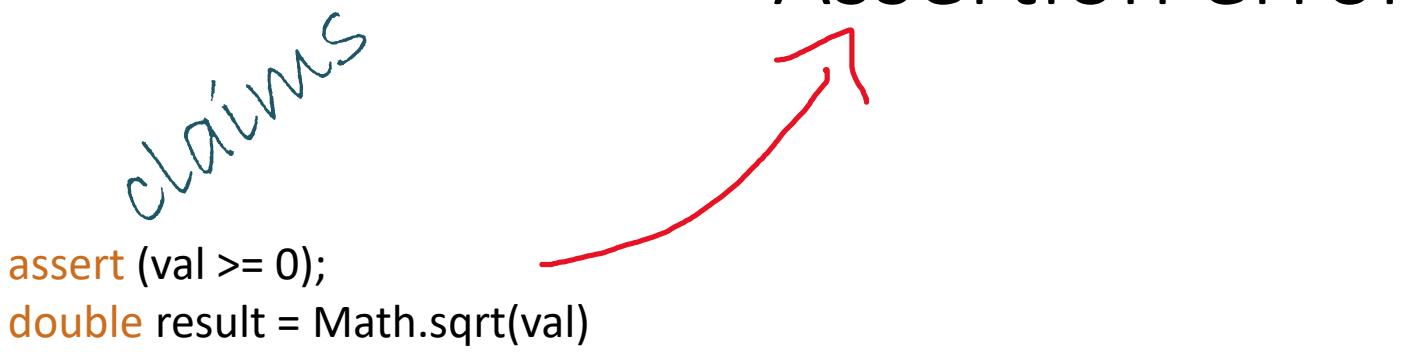
Assertions

Use ONLY when developing code

Finally we have to talk about "assertions". To "assert" means to state or to claim.
You claim that something is true. If it's wrong, the program crashes.
When you develop, it allows you to test that you are for instance getting values in the expected range.

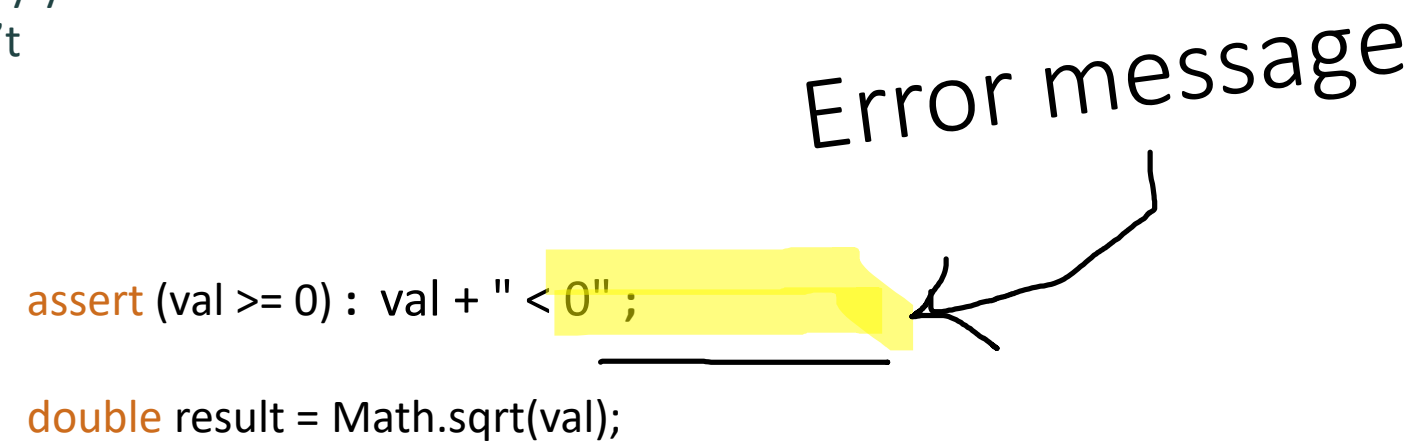
Assertions

Assertions allow you to check during testing that some exceptions won't occur.



Assertions

You can also add error messages, then track why you got a value you shouldn't



Assertions

- To use assertions:

  java –ea MyProgram

- Assertions are ignored if you don't pass the –ea flag (Enable Assertions) to java.
- They are a debugging tool.

Recursion

Searching

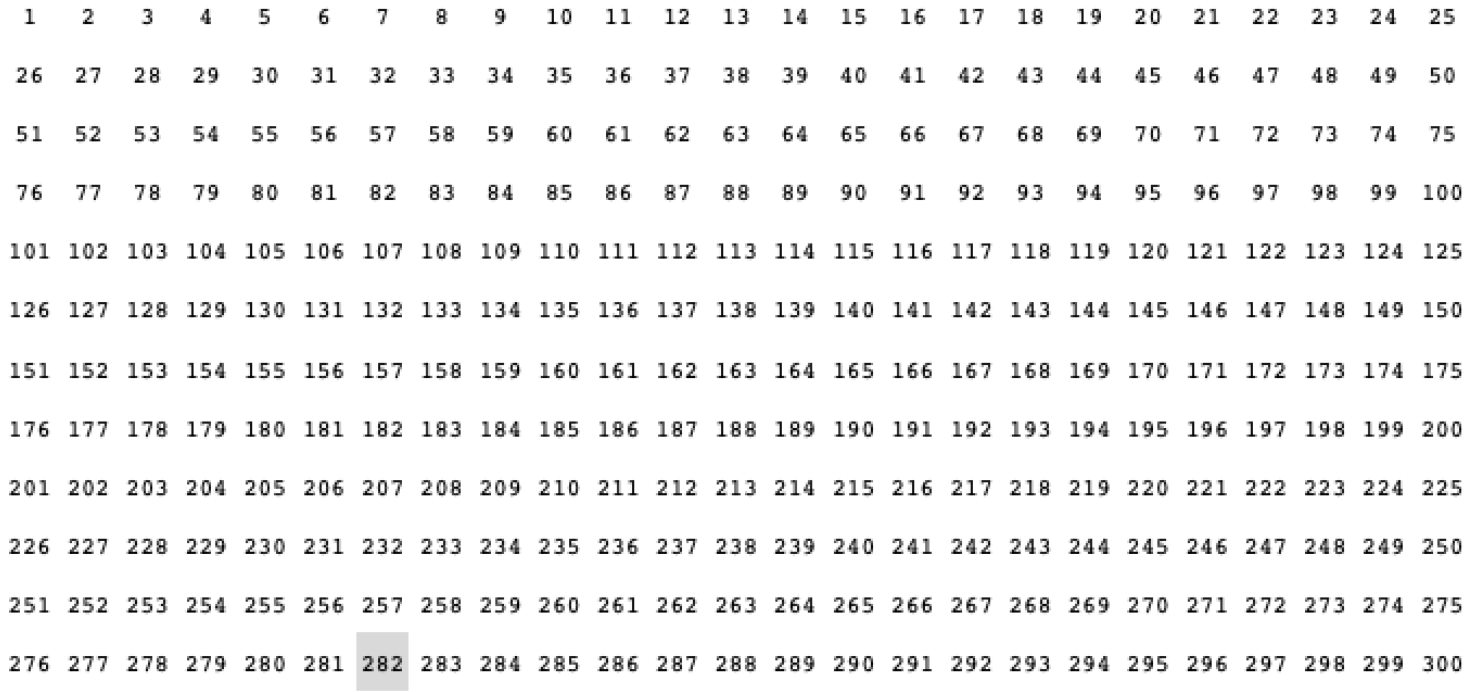
Guessing game

The computer randomly picks a number between 1 and 15, then you keep guessing numbers until you find the computers number. The computer will tell you each time if your guess was too high or too low.

Another Question

Guessing: binary search

The computer randomly picks a number between 1 and n, then you keep guessing numbers until you find the computers number. The computer will tell you each time if your guess was too high or too low.



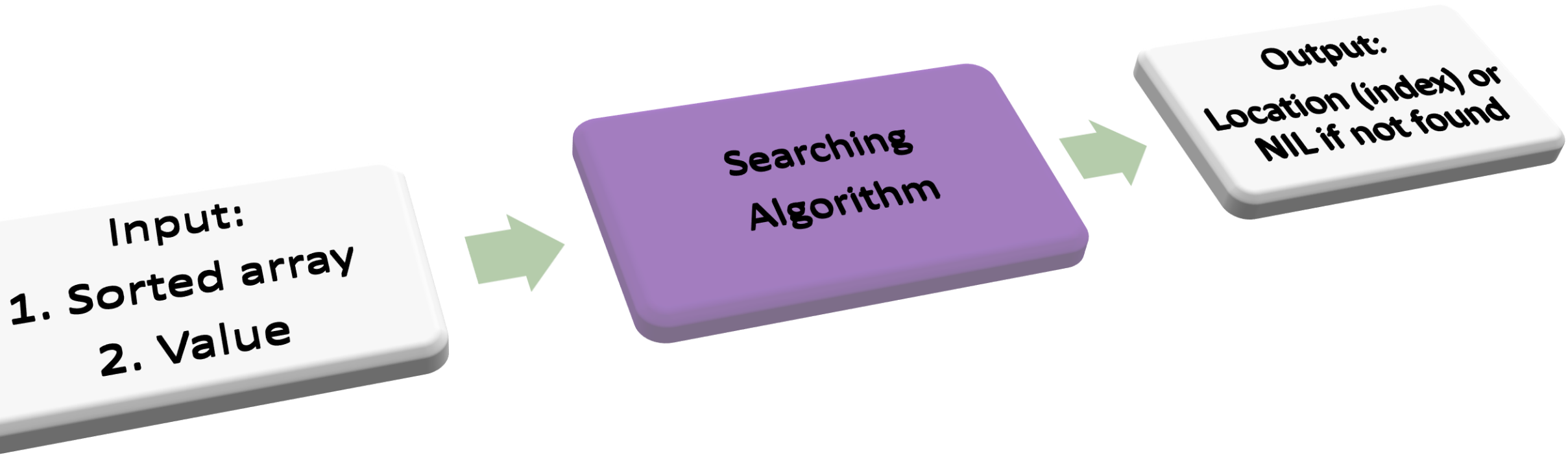
- But you could do something more efficient than just guessing 1, 2, 3, 4, …, right? Since the computer tells you whether a guess is too low, too high, or correct, you can start off by guessing the n/2.
- If the number that the computer selected is less than n/2, then because you know that n/2 is too high, you can eliminate all the numbers from n/2 to n from further consideration. If the number selected by the computer is greater than n/2, then you can eliminate 1 through n/2.
- Either way, you can eliminate half the numbers. On your next guess, eliminate half of the remaining numbers. Keep going, always eliminating half of the remaining numbers.

Searching problem

Instead of having an array with all integers in the range 1 to n included and guessing what number the computer had selected, we have an ordered array of integers (or other comparable items).

- **Input**: a sorted array A of integers and a value v
- **Output**: an index i such that A[i] = v or NIL if A does not contain the value v

Searching


Input:
1. Sorted array
2. Value
Searching Algorithm
Output:
Location (index) or NIL if not found

Iterative binary search

- Algorithm Parameters
  - Array A
  - Value v
  - low and high are array indexes
  - The procedure searches for v in the range A[low,…., high]

To search for v in an array call Iterative-binary-search(A,v,1,n] where n = A.length.

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
  while low ≤ high
      mid = ⌊(low + high)/2⌋
      if v == A[mid]
          return mid
      elseif v > A[mid]
          low = mid + 1
      else high = mid − 1
  return NIL
```

Recursive binary search

- Iterative and recursive version stop when the range is empty (low > high)
- If value v was found return the index (mid)
- If its not found try to continue the search with the range halved (either upper or lower half of the array)

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
  if low > high
      return NIL
  mid = ⌊(low + high)/2⌋
  if v == A[mid]
      return mid
  elseif v > A[mid]
      return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
  else return RECURSIVE-BINARY-SEARCH(A, v, low, mid − 1)
```

Alternative recursive binary search

- Exactly the same but we use some more variables i, j and p
- May be easier to see how it translates to java code with this version.

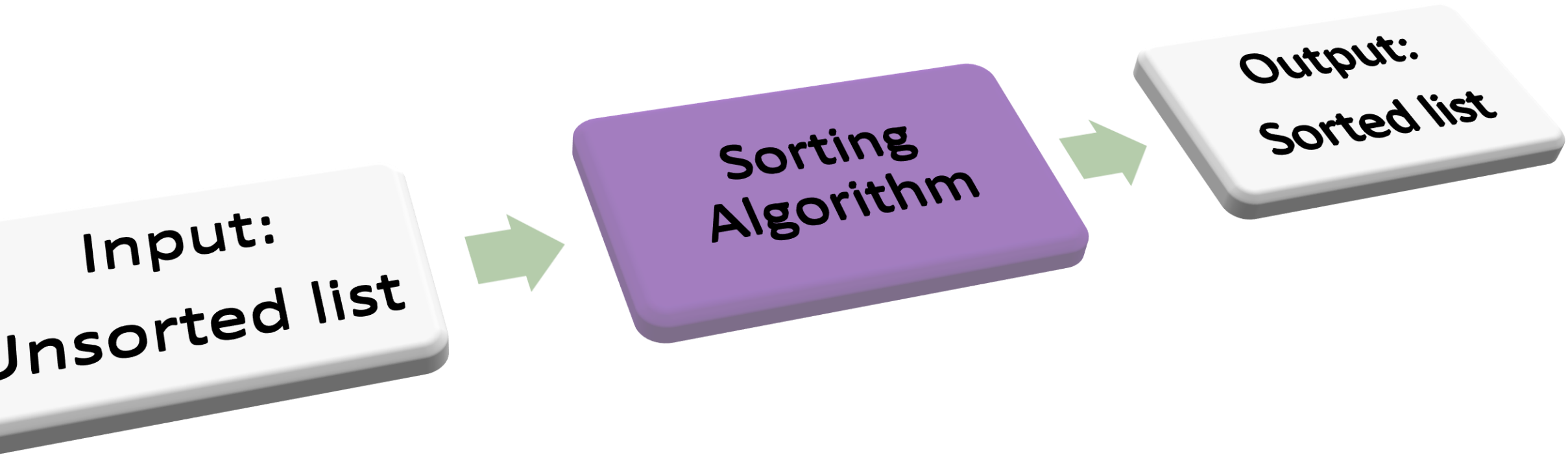
Algorithm 3 BINARY-SEARCH(A, v, p, r)

```
Input: A sorted array A and a value v.
Output: An index i such that v = A[i] or nil.
if p ⩾ r and v ≠ A[p] then
   return nil
end if
j ← A[⌊(r − p)/2⌋]
if v = A[j] then
   return j
else
   if v < A[j] then
      return BINARY-SEARCH(A, v, p, j)
   else
      return BINARY-SEARCH(A, v, j, r)
   end if
end if
```

Sorting

- Before computing sorting was a major bottleneck, slowing many operations, by requiring huge amounts of time. When the tabulating machine company (that later became IBM) was able to automate sort operations in the 1890s they sped up US Census data collection and reduced the costs dramatically.
- Computers mostly help us manage data; most programs, before they compute anything, have to retrieve data. And for retrieving data easily, data must be sorted, or kept in order. Sorting efficiently is a very important topic.

Sorting



Sorting algorithms

- Insertion sort
- Bubble sort
- Selection sort
- Quicksort
- Merge sort
- Heap sort

Insertion sort

```
function insertion_sort(list)
    for i ← 2 … list.length
        j ← i
        while j and list[j-1] > list[j]
            list.swap_items(j, j-1)
            j ← j - 1
```

6 5 3 1 8 7 2 4

Quick sort and Merge sort

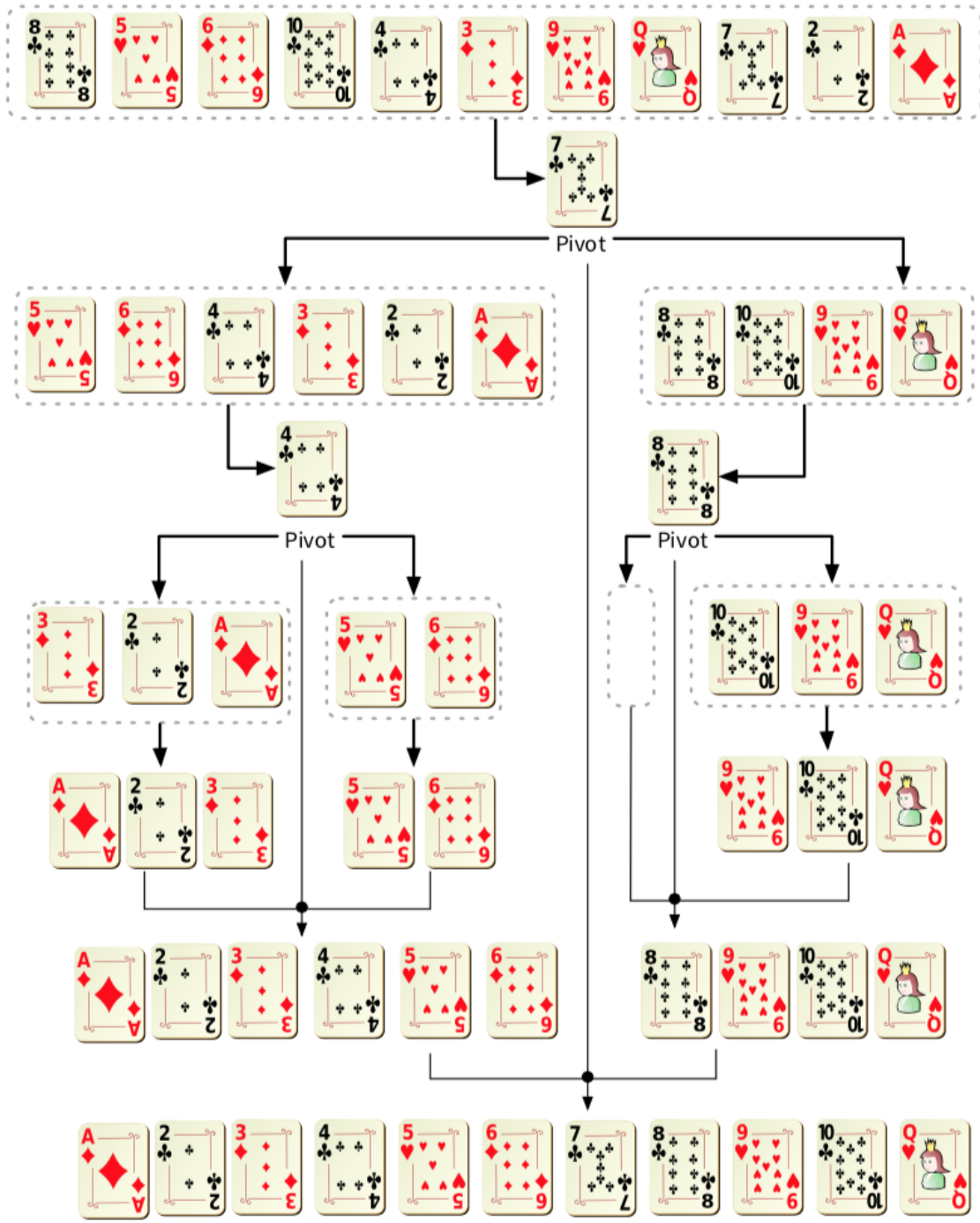
- For large lists that may not be nearly sorted it is necessary to use faster algorithms such as merge sort or quick sort
- These algorithms work by divide and conquer
  - Large problem is successively reduced to smaller sub problems until they are small enough to solve easily
  - Then the solution to the original problem is constructed from the pieces
  - Uses recursion

Quick sort

Using quick sort to sort a a pile of cards:

1. If the list has fewer than 4 cards, put them in the right order and finish, else go to step 2
2. Choose at random any card from the pile to be the pivot
3. Cards larger than the pivot go into a new pile to the right; cards smaller than the pivot go to the left
4. Start this procedure for each of the two piles to get a sorted pile

Exercise: get a deck of cards (or other sortable objects) and apply these steps to strengthen your understanding of quick sort.


Pivot
Pivot
Pivot

Remember

- There are many sorting algorithms
- Each is suitable for a specific situation although in the average case quick sort and merge sort are faster
- We will discuss what we mean by faster in the next presentation

Algorithm complexity

- Time complexity
- Space complexity

- Loops

```
void bubbleSort(int arr[])
{
    int n = array.length;
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (array[j] > array[j+1])
            {
                // swap
                int tmp = array[j];
                array[j] = array[j+1];
                array[j+1] = tmp;
            }
}
```

First pass:
n – 1 comparisons between elements
n/2 swaps (assume on average 50% out of order)

Second pass:
n-2 comparisons between elements
(n-1)/2 swaps (on average)

…

Time complexity in terms of operations performed:

Number of comparisons: **n(n-1)/2**

Number of swaps: **1⁄4 (*n* + 2)(*n* – 1)**

$O(n^2)$


BILL

O(1)

Means that the time is constant and doesn't depend on the number of objects processed. This is what happens when in a program you access the nth element in an array – whether you access the first or last element, time is the same, and doesn't depend either on the size of the array.

O(logn)

means that processing 1,000 times the number of objects will take about 3,000 more time. That's what the best sort algorithms can do.

O(n)

Means that the time is constant and doesn't depend on the number of objects processed. This is what happens when in a program you access the nth element in an array – whether you access the first or last element, time is the same, and doesn't depend either on the size of the array.

O(n logn)

Means that processing 1,000 times the number of objects will take about 3,000 more time. That's what the best sort algorithms can do.

$O(n^2)$

means that multiplying the number of values by 1,000 will multiply the time by 1,000,0000. That's what insertion sort and bubble sort typically do. You definitely don't want to use that on large numbers of values.

Selection sort

Insertion sort

$O(n^2)$

“Insertion sort", that first looks for the biggest value, then second biggest and so forth and looks more like what you might do by hand performs a *little* better than bubble sort, but is also an O(n2) sort. Yek!

O(n log n)

I have said to you, the best sorting algorithms are O(nlogn) algorithms. What does it mean compared to a O(n^2) algorithm?

“N log n” vs “n^2”:
Increases faster than n but not madly faster

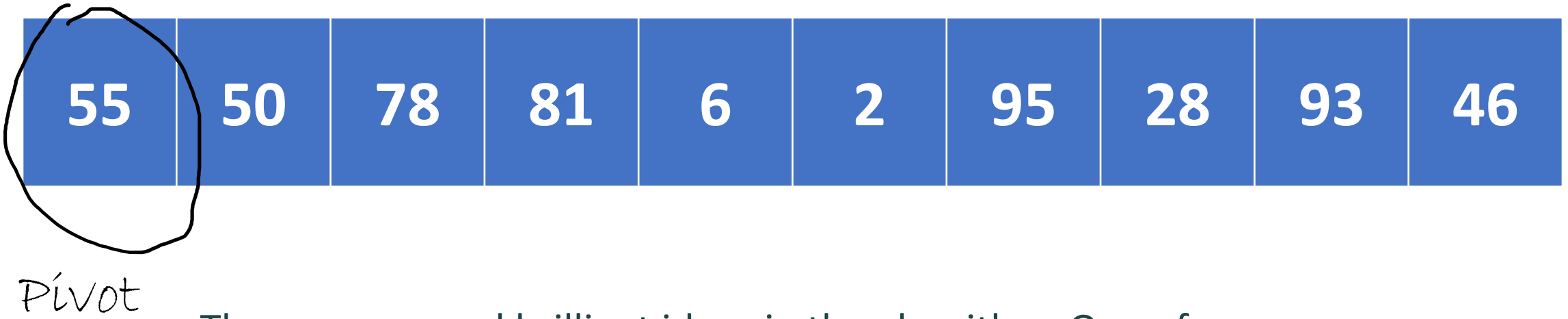
N	**N^2**	**N Log N**
10	100	10
100	10000	200
1000	1000000	3000

Quicksort

Antony Hoare (1934-)

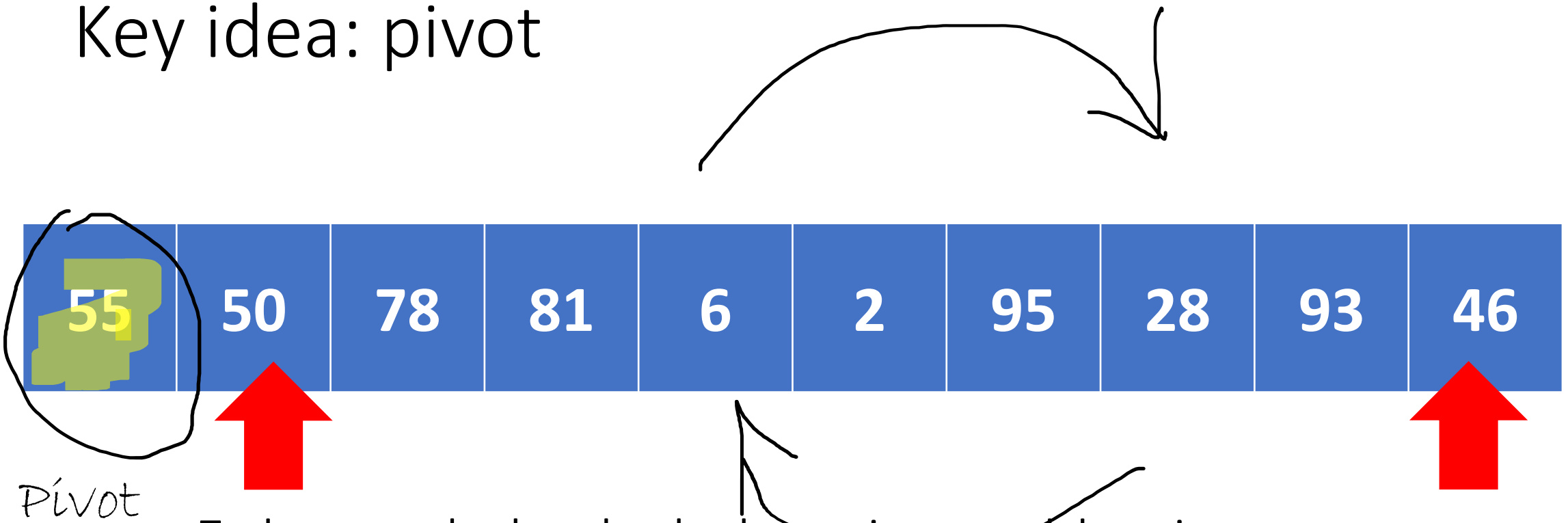
A first really fast sorting algorithm was invented by Antony Hoare in the early in Moscow. In 1959, while studying machine translation of languages in Moscow, he invented the now well-known sorting algorithm, "Quicksort."

Key idea: pivot



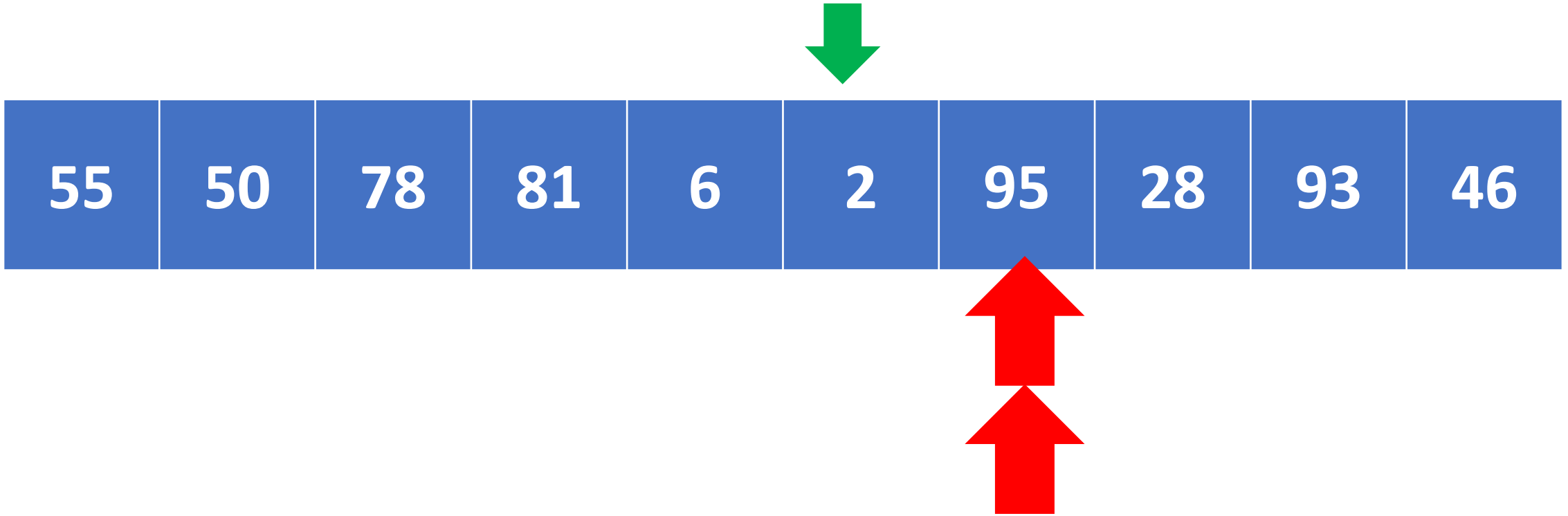
There are several brilliant ideas in the algorithm. One of them is, instead of successively looking for smallest (or greatest) values, to take arbitrarily one value called pivot and find its final location.

Key idea: pivot

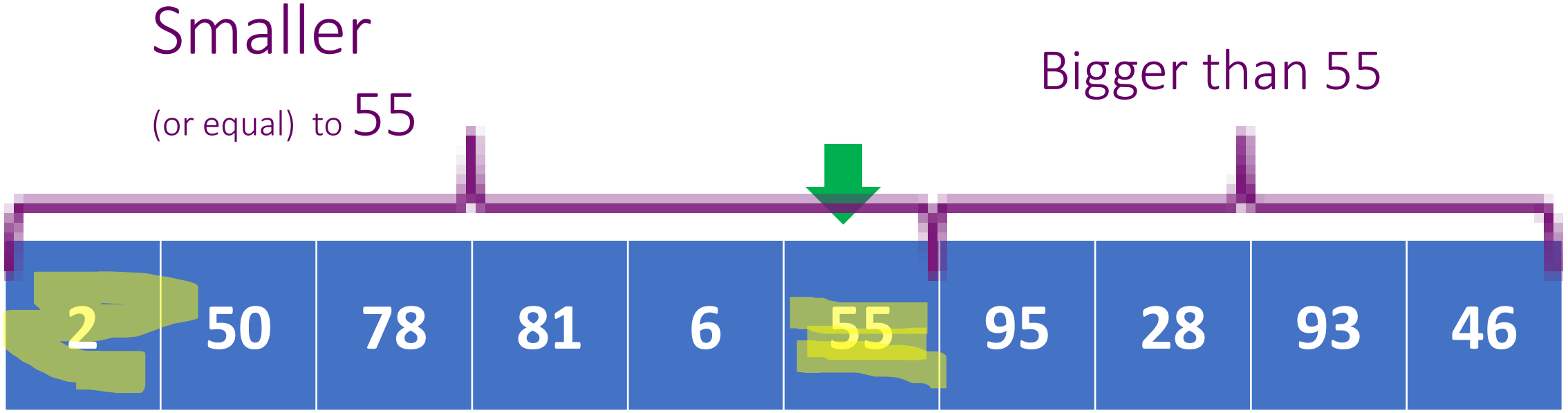


To do so, we check each value by moving up and down in the array at once, and stopping when the "up" pointer encounters a greater value than the pivot, and the "down" pointer a smaller one. Those values are swapped.

Key idea: pivot



When both pointers meet, everything on the right is bigger than the pivot, everything on the left is smaller (or equal) and we know that the final pivot location will be where the small green arrow points.



By swapping the pivot and the value occupying "its" place, we partition the original set into a subset containing smaller values, and a subset containing greater values. We "just" need to sort these two subsets.

KEY IDEA: “DIVIDE AND CONQUER”

Sorting twice the number of values is 4 times as costly

$$O(n^2)$$

As we have seen, most simple sorting algorithms have a cost in number of operations (and time) that increases as the square of the number of values sorted.

So it's better to perform two sorts applied to N values each than one sort applied to 2N values.

Java code to place pivot

Here is the Java code to find where the pivot goes: first we move two indices 'up' and 'down' until they find a

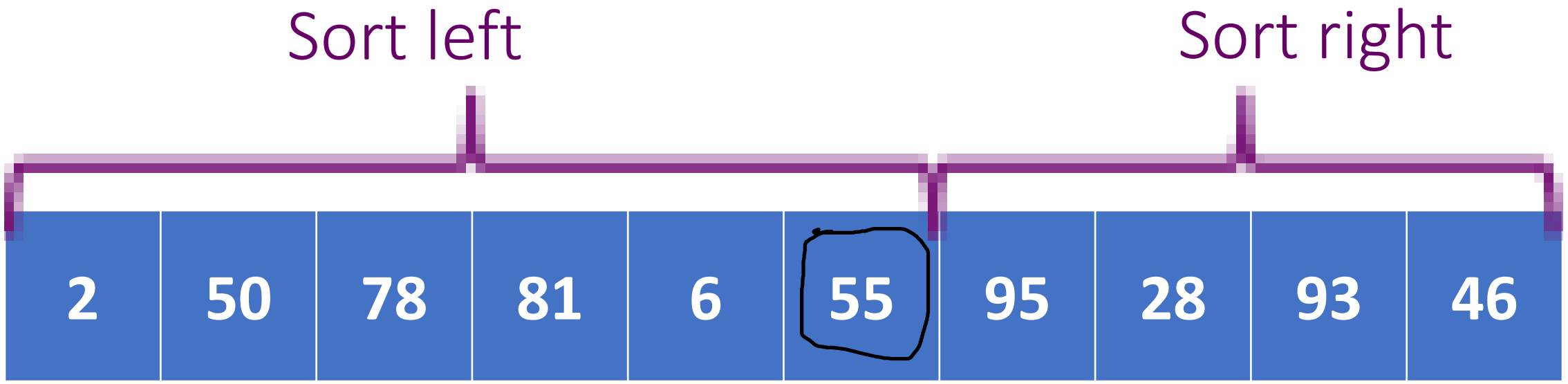
```
static int placePivot(int[] arr, int first, int last) {
    int pivot;
    int tmp;
    int up = first + 1; int down = last;

    pivot = arr[first]; while (down > up) {
        while ((arr[up] <= pivot) && (up < down)) {
            up++;
            }
            while ((arr[down] > pivot) && (up < down)) {
            down--;
            }
    }
}
```

Java code to place pivot

… then values are swapped. In the end, we return the pivot should go, which is the limit between the smaller subset and the bigger subset.

```
            if (up < down) {
                // Exchange values
                tmp = arr[up];
                arr[up] = arr[down];
                arr[down] = tmp;
            }
        }
        // up has stopped at a value > pivot
        // or when it has met the down pointer
        if  (pivot < arr[up]) {
            // Place pivot at up - 1
            up --;
        }
        arr[first] = arr[up];
        arr[up] = pivot;
        return up;
    }
```

The problem is that we must defer operations, sorting one subset, then the other, and if we apply each time the same "recipe" it can become very complicated.



"Remember" that we must sort 0 to 4 Sort 6 to 9

"Remember" that we must sort x to y

Sort w to Z

"Remember" ...

Another approach to sorting

- Bucket sort

Recursion

Mathematical Induction

- Mathematical induction is basically a very clever way of proving theorems that uses inductive reasoning

Mathematical induction

The sum of the n first odd integers is equal to n^2.

Proof outline

- - Obvious for 1
  - If it's true for n, it's true for n+1
- Therefore it's true for any integer value.

Mathematical induction

The sum of the n first odd integers is equal to n^2.

Proof

Base case: It is obvious for 1 (ie 1^2 = 1).

Inductive step: Suppose it is true for n. Then,

1 + 3 + ... + (2n – 1) = n^2.

We will try to show that if the statement is true for n, then it follows from that it is true for n+1

1 + 3 + ... + (2n – 1) + (2n + 1)

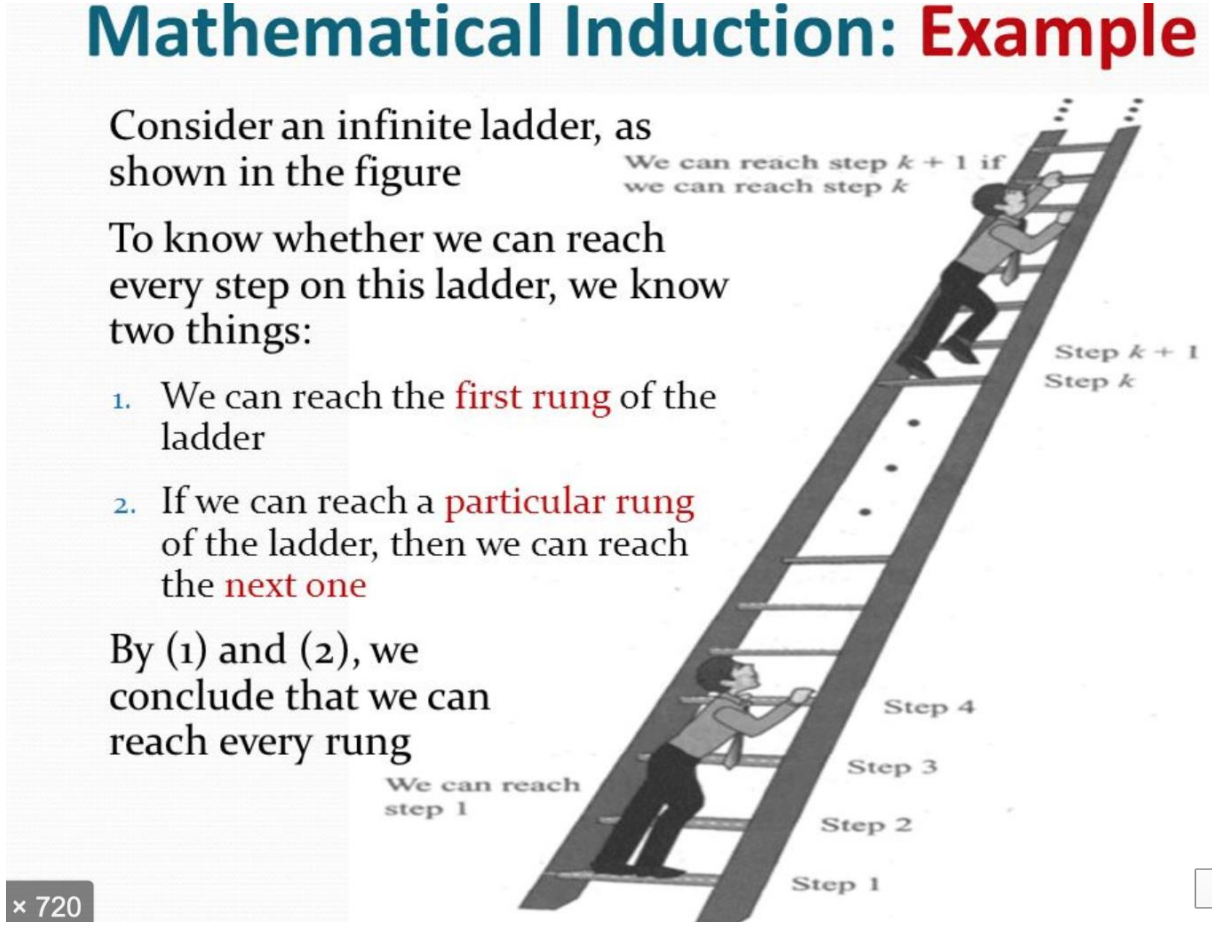
Next odd integer after the nth one

Substituting from the inductive step gives: n^2 +(2n+1) = (n+1)^2 [by factoring n^2 + 2n +1]

QED

How Mathematical Induction works

- Base case
- Link between *n* and *n + 1* (the truth of n+1 follows from the truth of n)

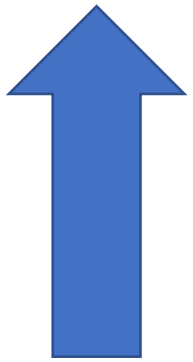


Recursion

- Is a related approach in programming
- Is also based on a link between n and n+ 1 and a trivial case, but it works in reverse order
- We often use mathematical induction to prove recursive algorithms work. Instead of having an infinite number of cases we look at the boundary conditions of the loop.

Mathematical Induction

- Link between n and n+1
- Base case



Mathematical induction goes from the trivial case towards infinity.

Recursion

- Link between n and n+1
- Base case

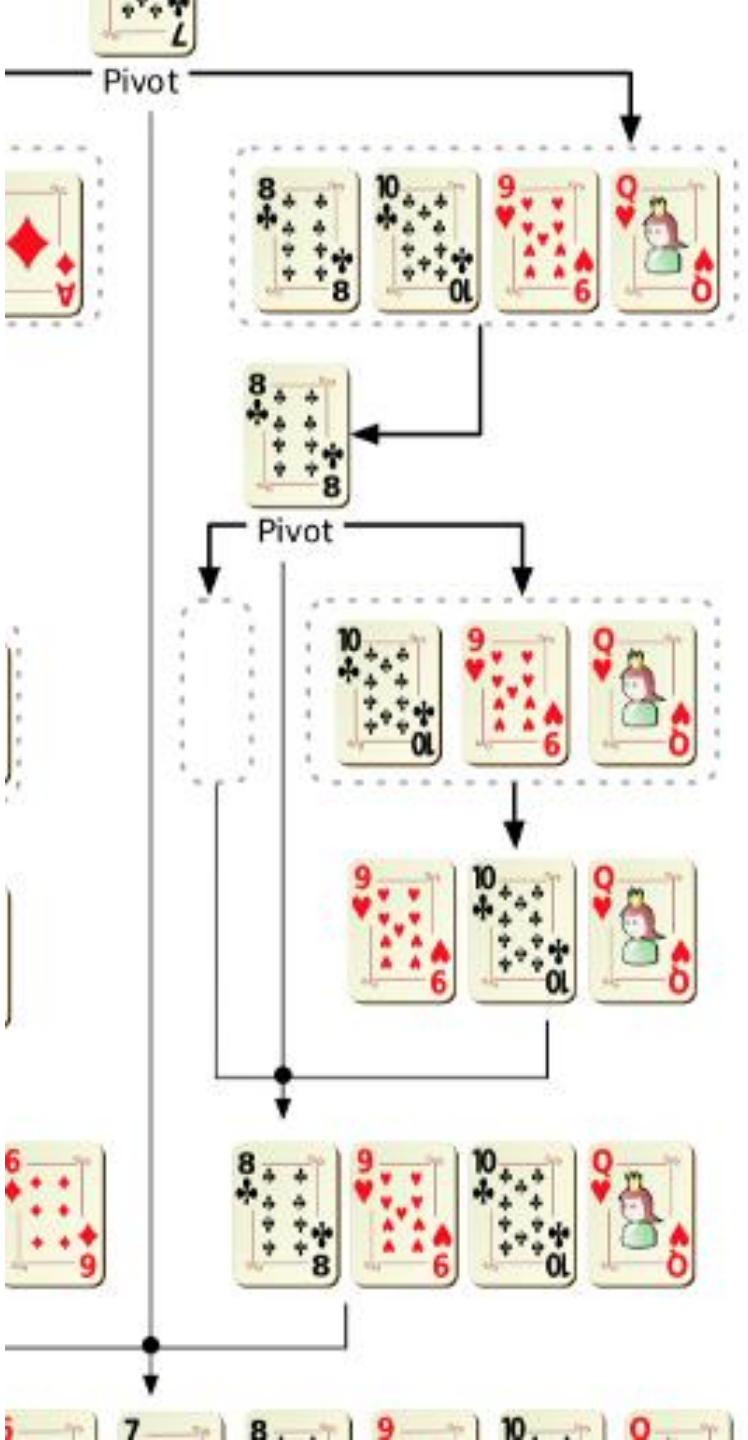
Recursion, which we'll use for the Quick-Sort, works by identifying a trivial case, then assuming that we can solve a problem at level n-1 and expressing the solution to the problem at level n as a function of the n-1 level solution. Once the trivial case is found it works the solution back to level n.

Recursion

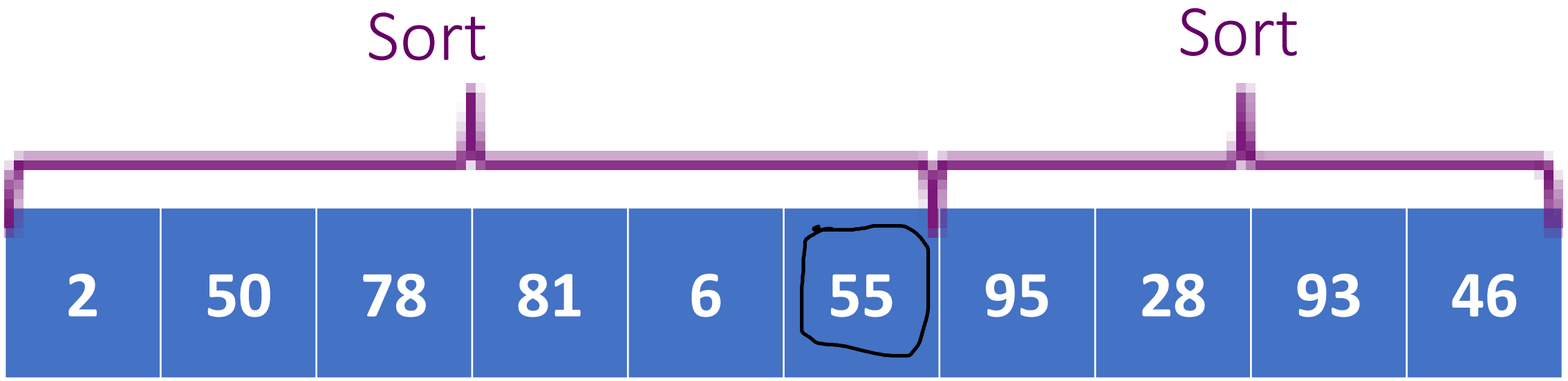
Def: A function contains a call to itself (with other parameters).

Recursion is characterized by functions that contain calls to themselves, with different parameters corresponding to a smaller problem.

- Of course at some point the function must reach a very easy case and no longer call itself! There will necessarily be a condition in the function to stop the recursion.
- We also call this the base case.
- With cards we said this was the case with *less than* 4 cards.



With a recursive function you need to define a base case…




```
static void quickSort(int[] arr, int first, int
last) {
```

BASE CASE: arrays with 1 value are easy to sort

RECURSIVE QUICKSORT IMPLEMENTATION

- If there is 0 or 1 value in the array, nothing to do. If there are two values in the array we just check that they are in order and swap them if not.

```
static void quickSort(int[] arr, int first, int last) {
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first) {
        case 1: // base case
            if (arr[last] < arr[first]) {
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
```

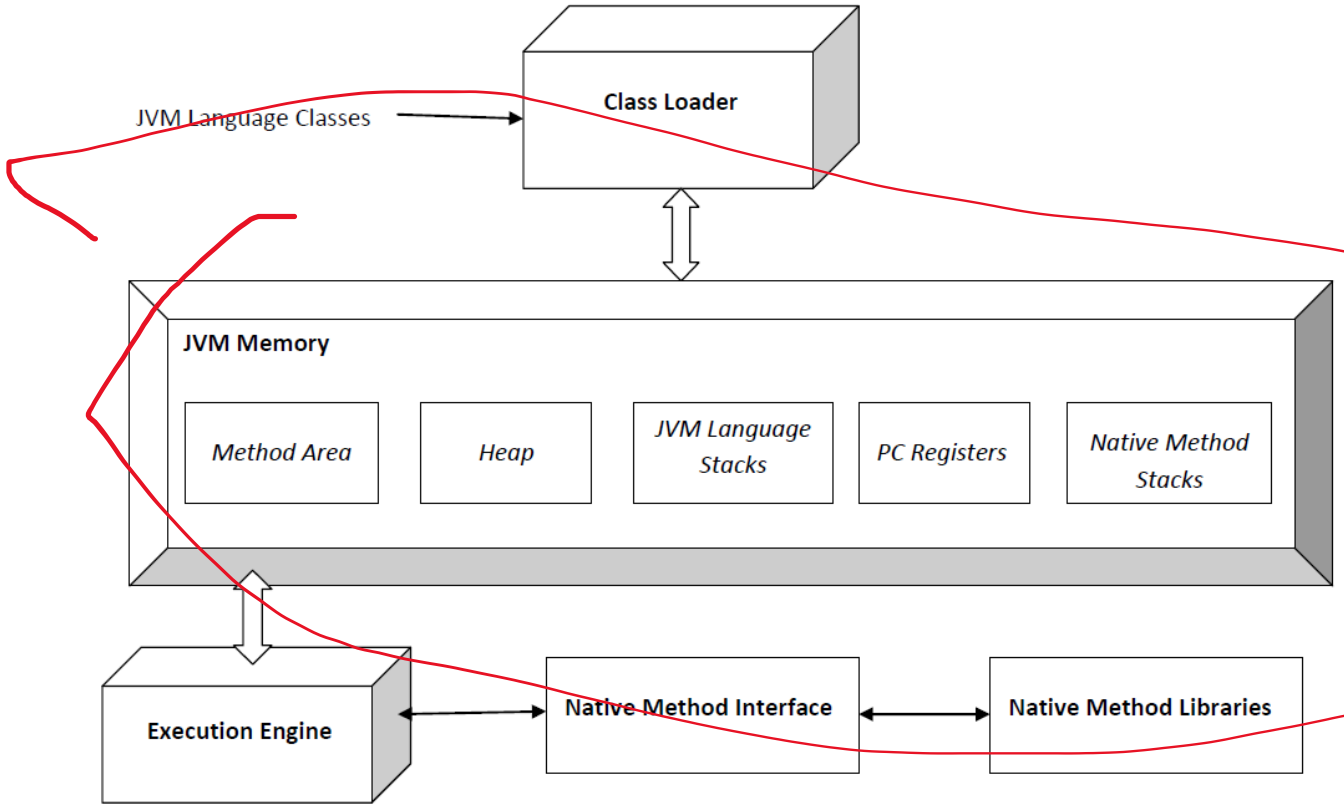
RECURSIVE QUICKSORT IMPLEMENTATION

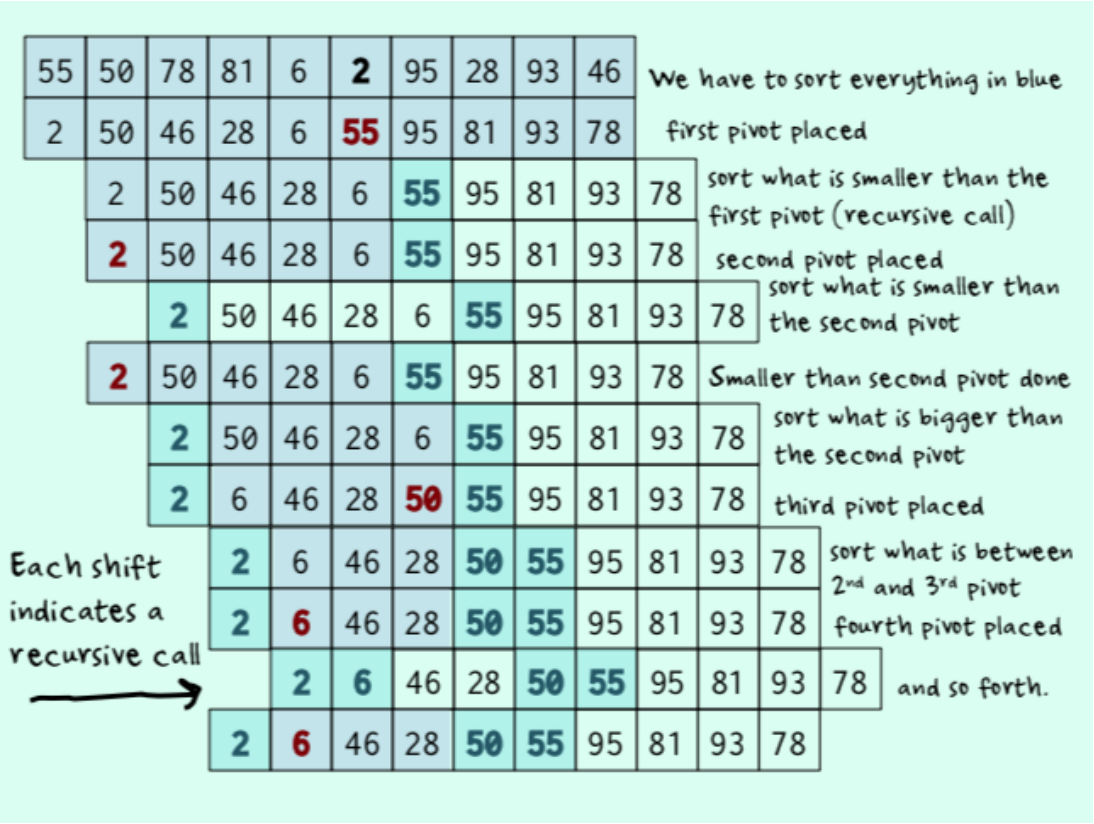
In the general case, we place the pivot, then call the function again for sorting the two subsets of the original array.

And THAT'S ALL!

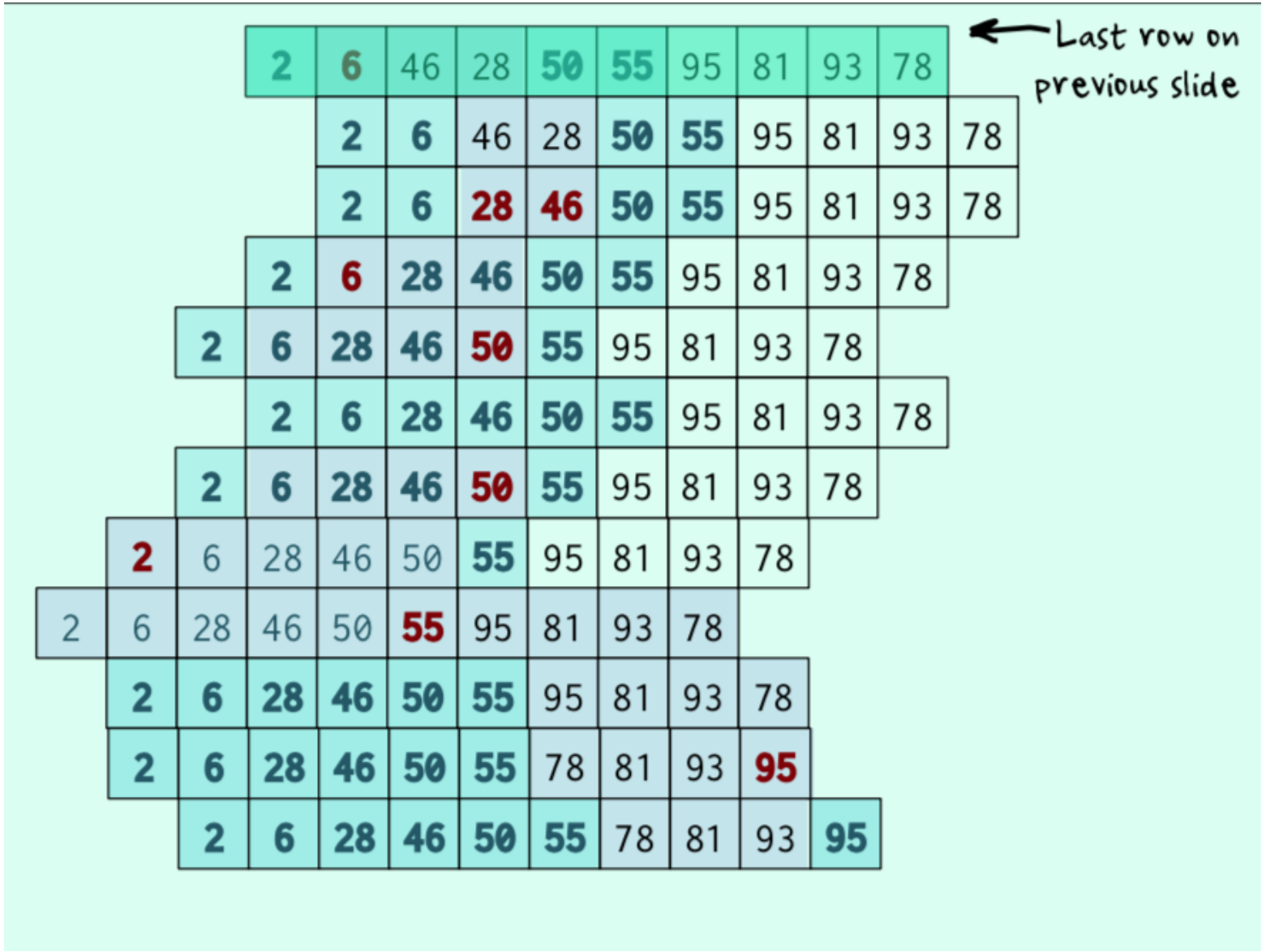
```
static void quickSort(int[] arr,
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first)
        case 1: // base case
            if (arr[last] < arr[
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
        default:
            pivotPos = placePivot(arr, first, last);
            quickSort(arr, first, pivotPos-1); // sort left
            quickSort(arr, pivotPos+1, last); // sort right
            break;
            }
    }
}
```

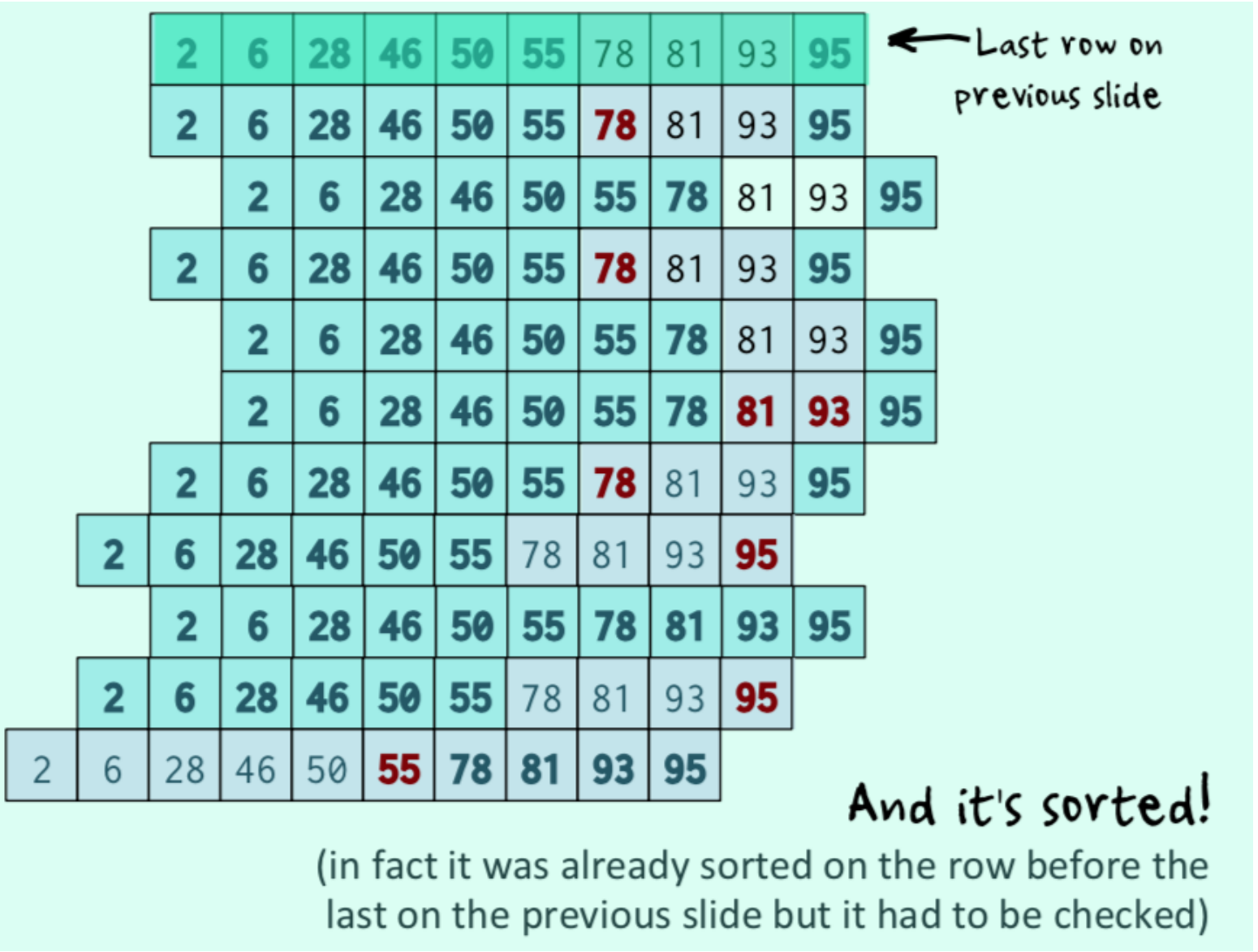
The algorithm can be improved (clever choice of pivot, sorting what is smaller
first, etc ...).


JVM Language Classes
Class Loader
JVM Memory
Method Area
Heap
JVM Language Stacks
PC Registers
Native Method Stacks
Execution Engine
Native Method Interface
Native Method Libraries



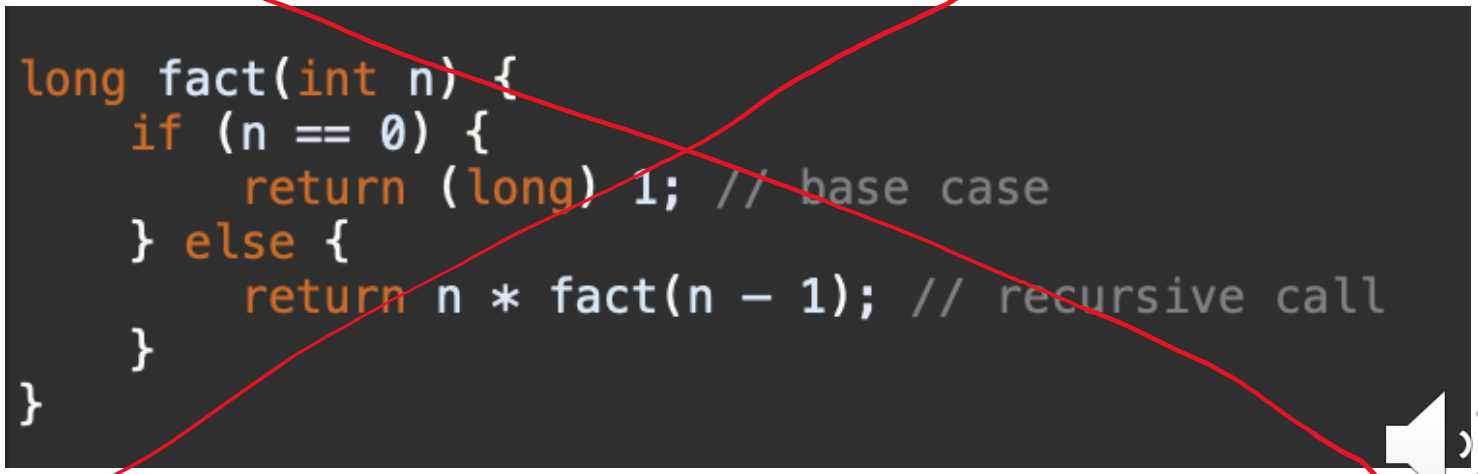

```java
static void quickSort(int[] arr, int first, int last) {
    int pivotPos;

    int tmp;
    if (last > first) {
        switch (last - first) {
        case 1: // base case
            if (arr[last] < arr[first]) {
                tmp = arr[last];
                arr[last] = arr[first];
                arr[first] = tmp;
            } break;
        default:
            pivotPos = placePivot(arr, first, last);
            quickSort(arr, first, pivotPos-1); // sort left
            quickSort(arr, pivotPos+1, last); // sort right
            break;
            }
    }
}
```


Last row on previous slide
2 6 46 28 50 55 95 81 93 78
2 6 46 28 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 95 81 93 78
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95


2 6 28 46 50 55 78 81 93 95
Last row on previous slide
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
2 6 28 46 50 55 78 81 93 95
And it's sorted!
(in fact it was already sorted on the row before the last on the previous slide but it had to be checked)

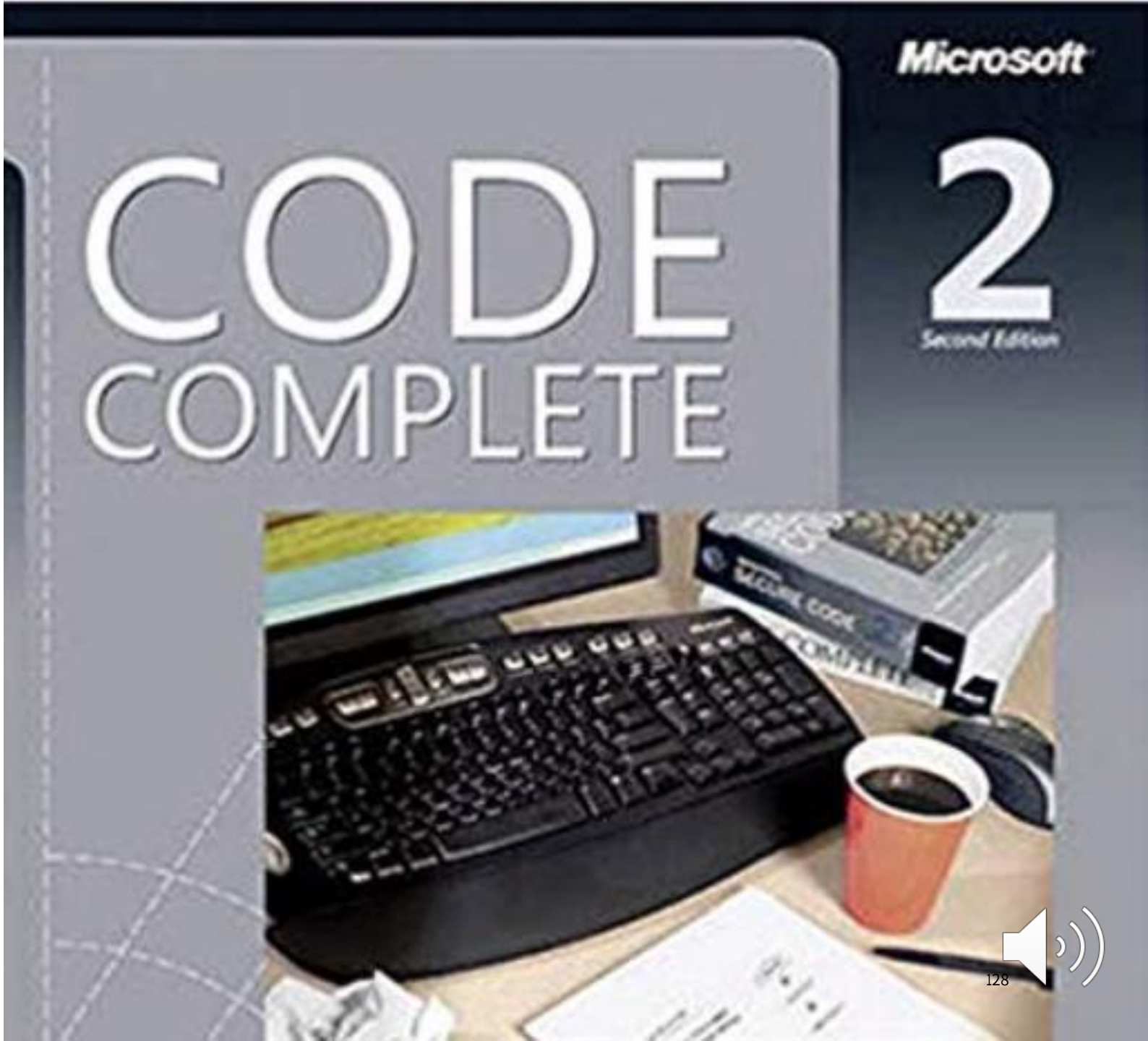
"One problem with computer-science textbooks is that they present silly examples of recursion. The typical examples are computing a factorial or computing a Fibonacci sequence. Recursion is a powerful tool, and it's really dumb to use it in either of those cases. If a programmer who worked for me used recursion to compute a factorial, I'd hire someone else. . . In addition to being slow and making the use of run-time memory unpredictable, the recursive version of [a factorial-computing] routine is harder to understand than the iterative version . . ."

--- Code Complete Steve Carrol



Software Design Practices

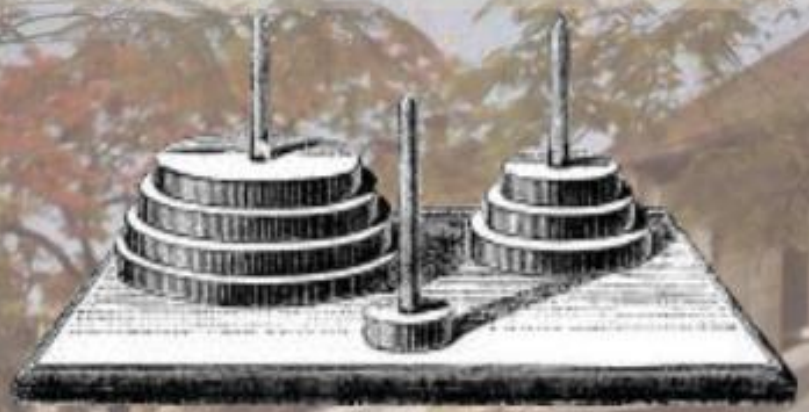
- Code Complete
- Often recommended book about software practice
- Bit old but probably a good source to understand software design practices



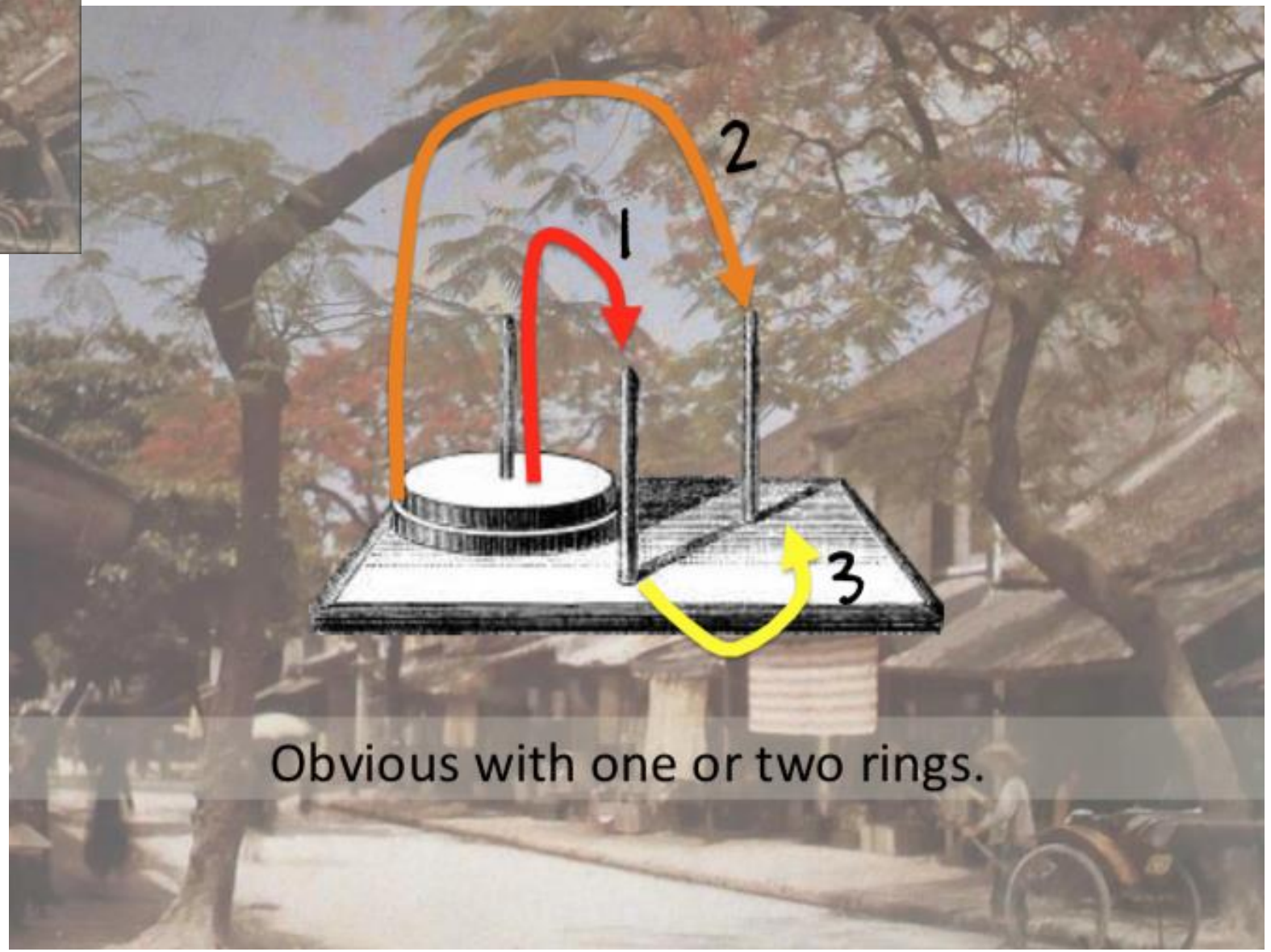
```java
long fact(int n) {
    if (n == 0) {
        return (long) 1; // base case
    } else {
        return n * fact(n - 1); // recursive call
    }
}
```

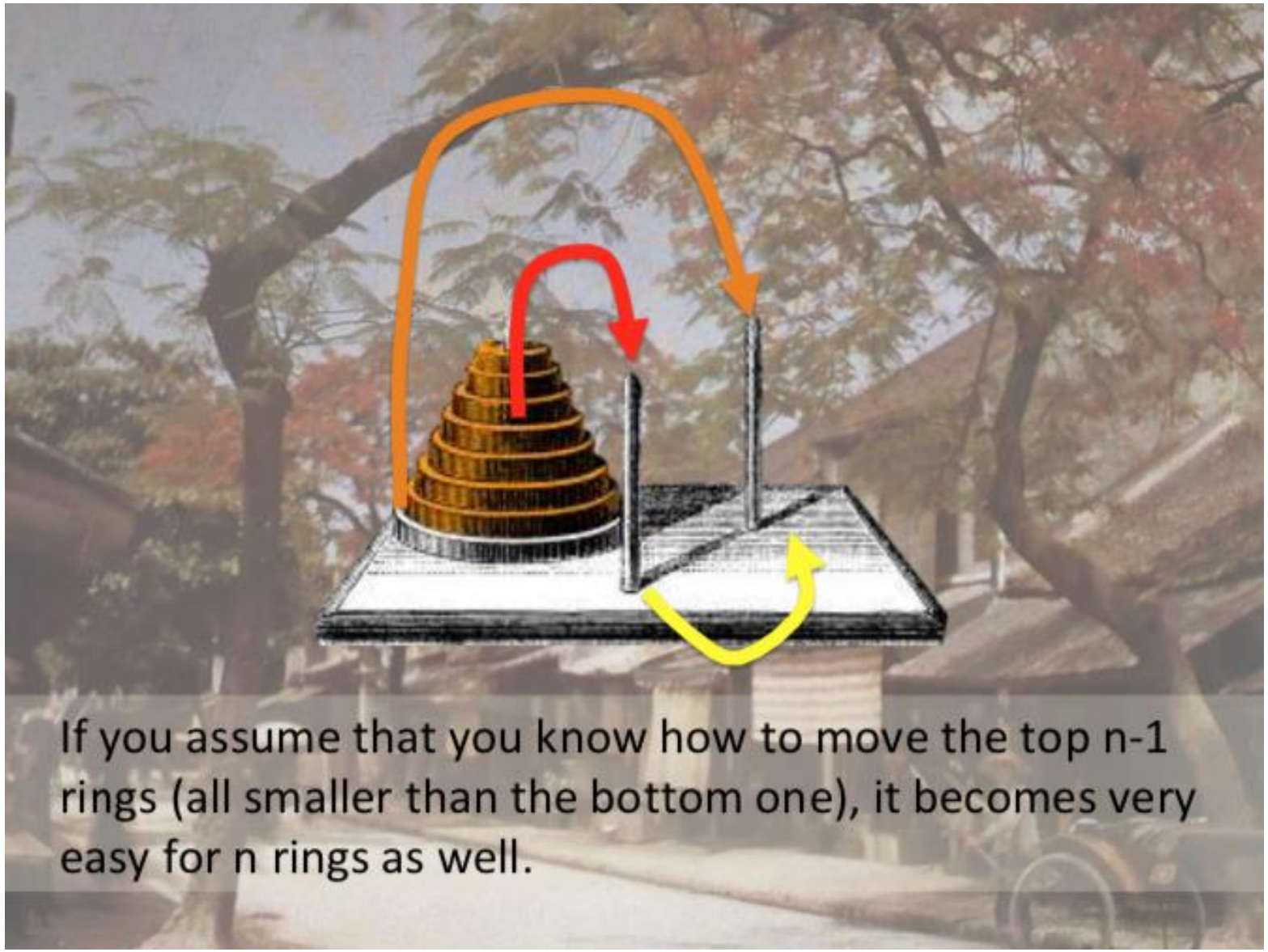
```java
int factorial(int n) {
    int f = 1;
    for(int i=1;i<=n;i++) {
        f *= i;
    }
    return f;
}
```

Another famous (but good) recursion example popular with textbooks is the towers of Hanoi problem, a puzzle invented by a French mathematician in the late 19th century.



The goal is to move a stack of discs or rings of decreasing radius from one peg to another, using an intermediary third peg, never stacking a bigger ring over a smaller one.


2
1
3
Obvious with one or two rings.


If you assume that you know how to move the top n-1 rings (all smaller than the bottom one), it becomes very easy for n rings as well.

Recursive Algorithm for Towers of Hanoi Problem

```
move_tower(tower_size, from_peg, to_peg, using_peg)
      if tower_size is 1
            move ring from from_peg to to_peg
      else if tower_size is 2
            move top ring from from_peg to using_peg
            move bottom ring from from_peg to to_peg
            move top_ring from using_peg to to_peg
      else
            move_tower(tower_size -1, from_peg, using_peg,
to_peg)
            move bottom ring from from_peg to to_peg
            move_tower(tower_size - 1, using_peg, to_peg,
from_peg)
```

Towers of Hanoi: 5 rings