CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn

As we have seen last time, 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

You can also have logic errors, either in your application (Java won't see it) or in your coding (Java will see it).

Compile-time
Link-time
Run-time
Logic

Program terminates with a wrong result

Program crashes

Divide by zero
Computed array index out of bounds
Null object reference

These errors and runtime exceptions aren't expected to happen, and are called "unchecked exceptions".

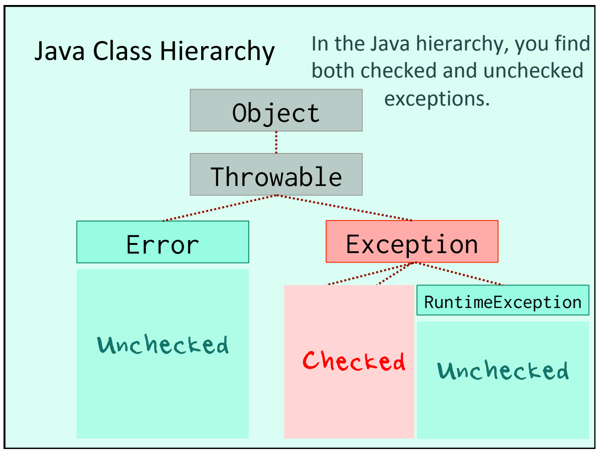
UNCHECKED EXCEPTION

Errors

Runtime Exceptions

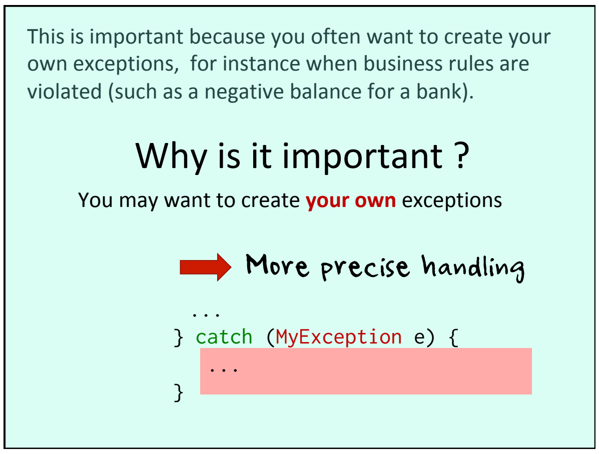
No requirement for catching or declaring

Rules are relaxed for them.



Java Class Hierarchy

In the Java hierarchy, you find both checked and unchecked exceptions.



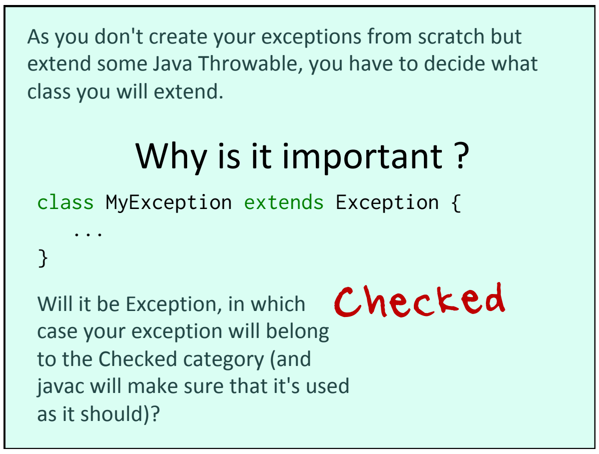
This is important because you often want to create your own exceptions, for instance when business rules are violated (such as a negative balance for a bank).

Why is it important ?

You may want to create **your own** exceptions

More precise handling

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



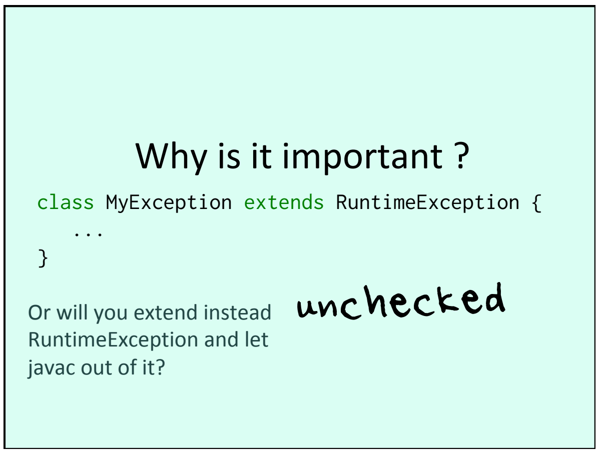
As you don't create your exceptions from scratch but extend some Java Throwable, you have to decide what class you will extend.

Why is it important ?

```
class MyException extends Exception {
    ...
}
```

Will it be Exception, in which case your exception will belong to the Checked category (and javac will make sure that it's used as it should)?

Checked



Why is it important ?

```
class MyException extends RuntimeException {
    ...
}
```

Or will you extend instead RuntimeException and let javac out of it?

unchecked

In principle, it all depends on how often you expect things to go wrong (and a big bank, "once in a million" in a transaction may mean "every two days")

Exception

"Exceptional Event"

Unlikely?

Once a week?

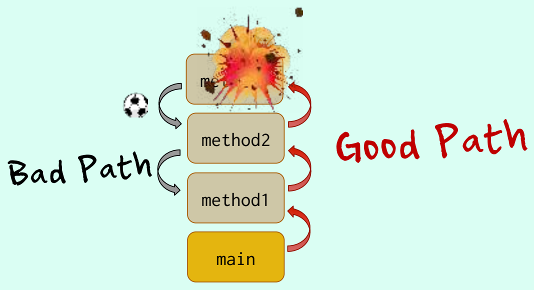
Not supposed to happen?

Once every five years?

Checked is safer

In theory, checked is better as javac will ensure that all the users of your method work properly. Practice 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.

Processing Exceptions



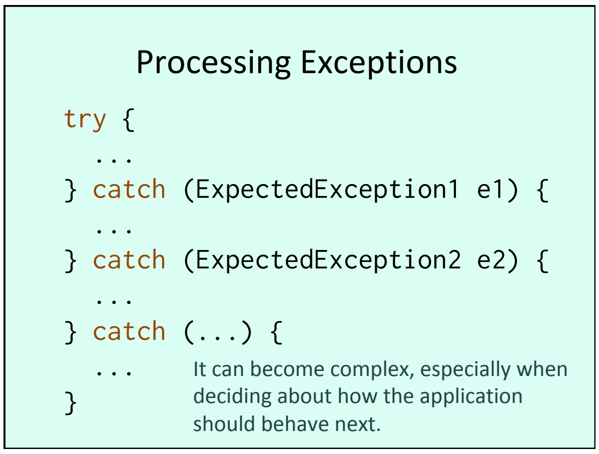
Whether your exception is checked or unchecked, you have to take into account what happens when it's thrown.

Processing Exceptions

```
try {

    ...


} catch (ExceptionTypeName e) {

    ...

}
```

You have the "everything is fine" case

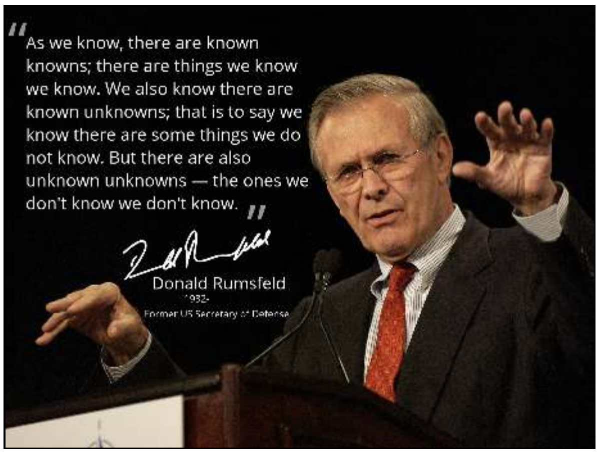
and if you catch the exception you need to thing about what to think about do when things turn bad as they often do.



Processing Exceptions

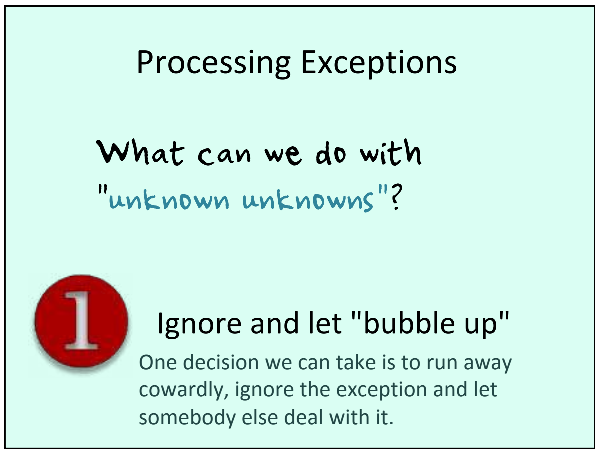
```
try {
  ...
} catch (ExpectedException1 e1) {
  ...
} catch (ExpectedException2 e2) {
  ...
} catch (...) {
  ...
}
```

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



"As we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns — the ones we don't know we don't know."

Donald Rumsfeld
1932-
Former US Secretary of Defense

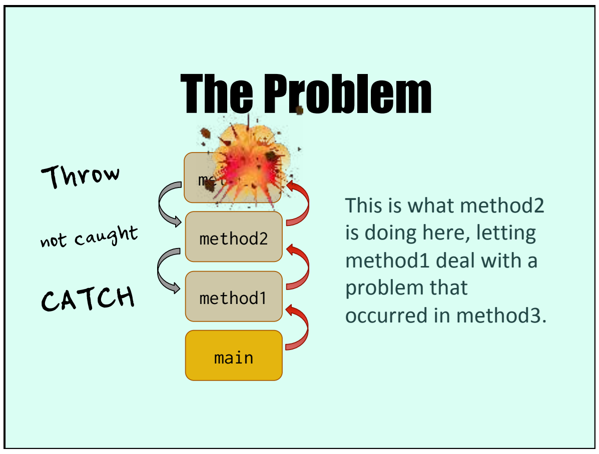


Processing Exceptions

What can we do with "unknown unknowns"?

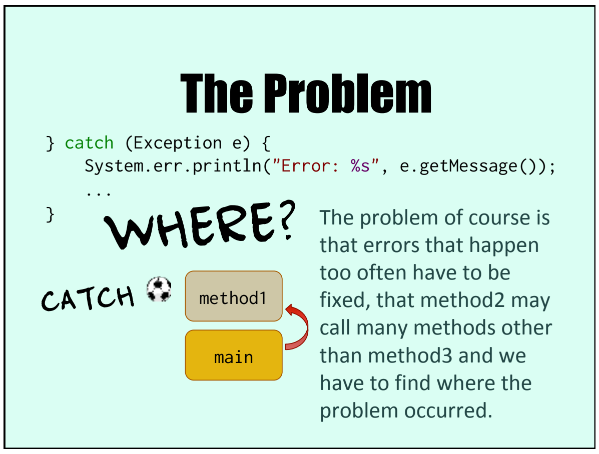
1 Ignore and let "bubble up"

One decision we can take is to run away cowardly, ignore the exception and let somebody else deal with it.



The Problem

This is what method2 is doing here, letting method1 deal with a problem that occurred in method3.



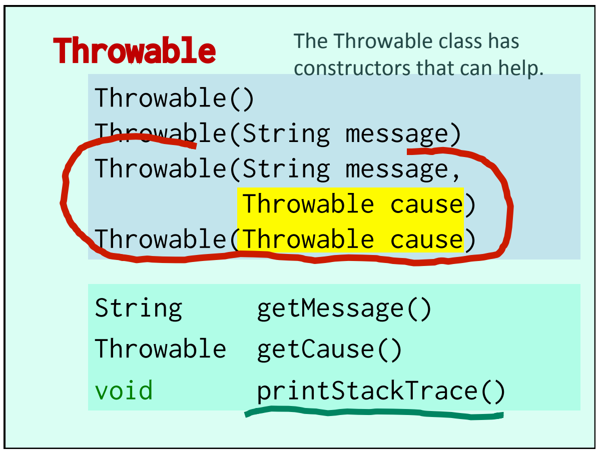
The Problem

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

WHERE?

CATCH
method1
main

The problem of course is that errors that happen too often have to be fixed, that method2 may call many methods other than method3 and we have to find where the problem occurred.

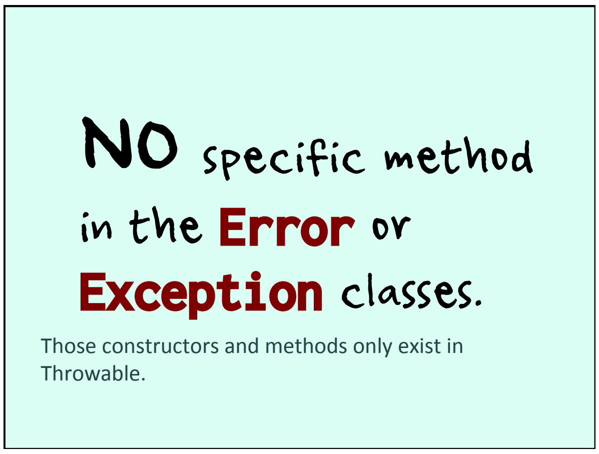


Throwable

The Throwable class has constructors that can help.

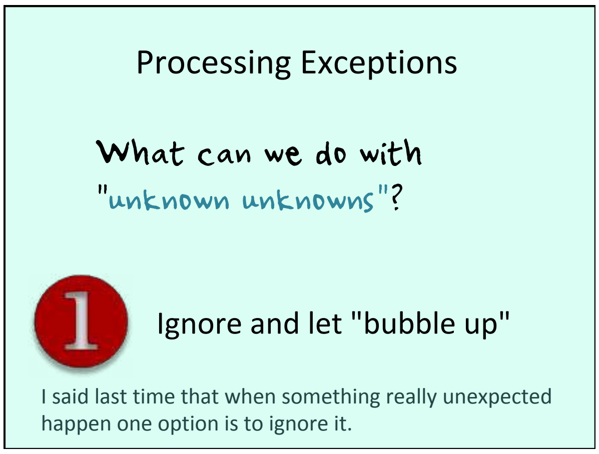
```
Throwable()
Throwable(String message)
Throwable(String message,
          Throwable cause)
Throwable(Throwable cause)
```

```
String      getMessage()
Throwable   getCause()
void        printStackTrace()
```



NO specific method in the **Error** or **Exception** classes.

Those constructors and methods only exist in Throwable.

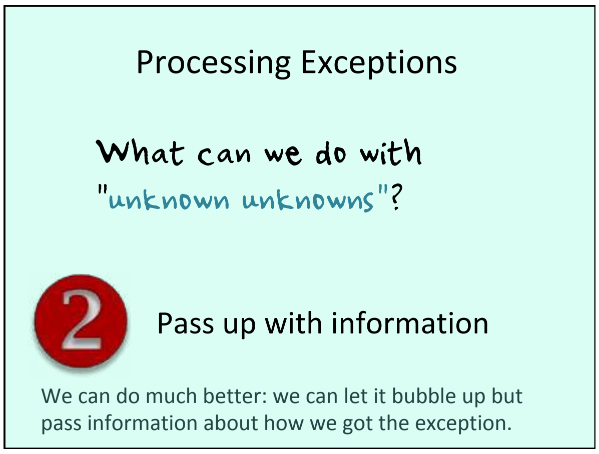


Processing Exceptions

What can we do with "unknown unknowns"?

1 Ignore and let "bubble up"

I said last time that when something really unexpected happen one option is to ignore it.

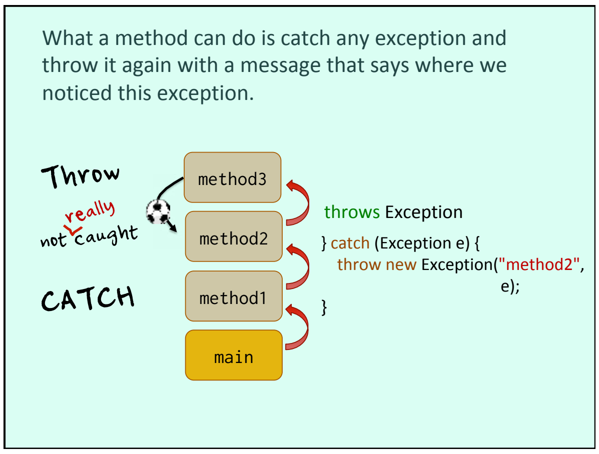


Processing Exceptions

What can we do with "unknown unknowns"?

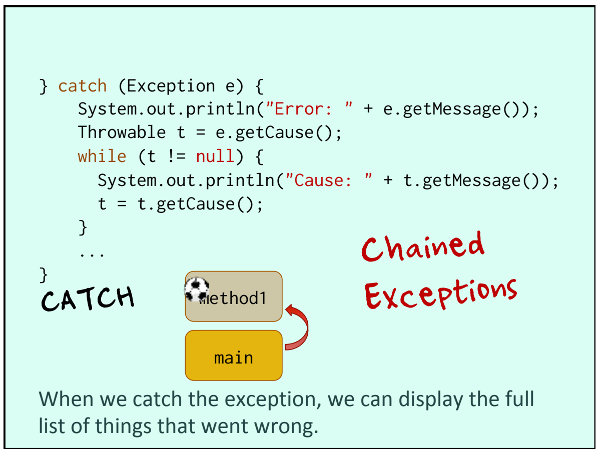
2 Pass up with information

We can do much better: we can let it bubble up but pass information about how we got the exception.



What a method can do is catch any exception and throw it again with a message that says where we noticed this exception.

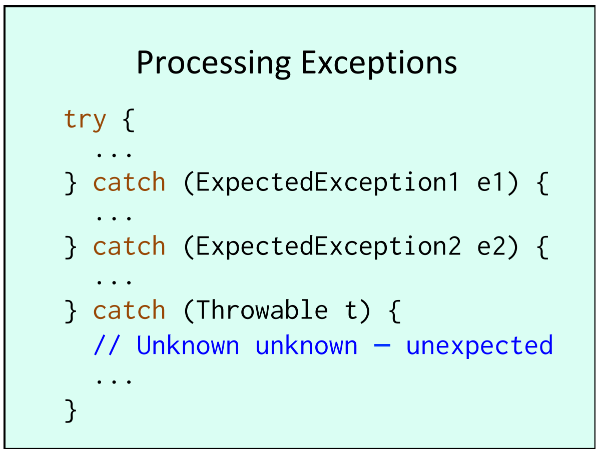
```
throws Exception
} catch (Exception e) {
   throw new Exception("method2",
                       e);
}
```



```
} 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();
    }
    ...
}
```

CATCH
method1
main
Chained Exceptions

When we catch the exception, we can display the full list of things that went wrong.



Processing Exceptions

```
try {
   ...
} catch (ExpectedException1 e1) {
   ...
} catch (ExpectedException2 e2) {
   ...
} catch (Throwable t) {
   // Unknown unknown – unexpected
   ...
}
```

You can 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 ...

There is a delicate balance between writing a program that your grand-mother can use ...

Don't SCARE end-users with heavily technical 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)
$
```

But provide enough info for the support team

~~A technical problem occurred.~~
~~Please contact support.~~

... 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.

Some operations need to be performed in all cases

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

}
```

In some cases, you need to run some tasks whether the whole operations worked or not, such as closing a connection if opening it was successful.

Some operations need to be performed in all cases

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

Some operations need to be performed in all cases

New alternative syntax (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.

Declaration(s) must implement the (auto)closeable interface

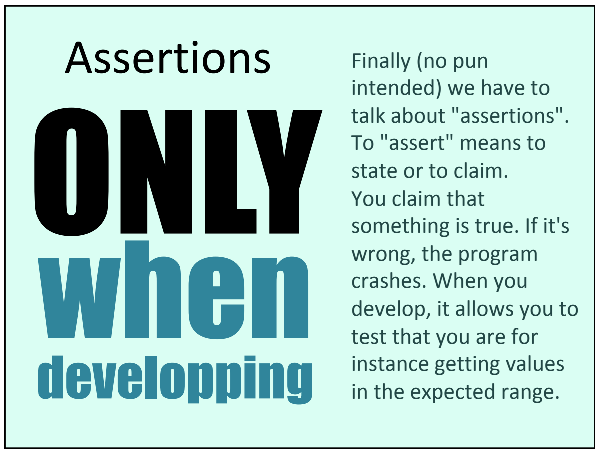
```
try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next()) {
        ....
    }
} catch (SQLException e) {
    ...
}
```

call .close() when exiting the try block

Example taken from the docs

Same as

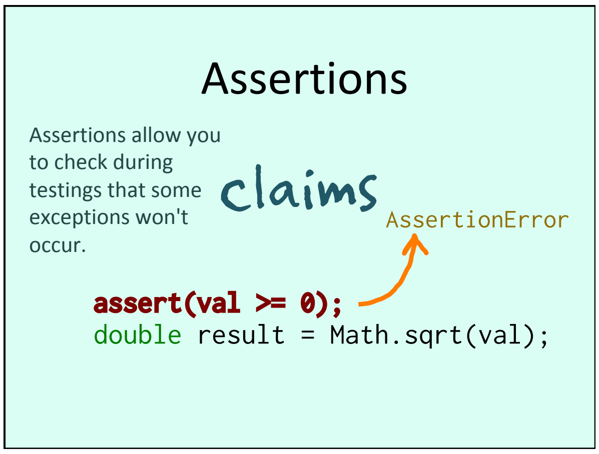
```
Statement stmt = null;
try {
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next()) {
        ....
    }
} catch (SQLException e) {
    ...
} finally {
  try {
    stmt.close();
  } catch (SQLException e) { // Do nothing
  }
}
```



Assertions

ONLY when developping

Finally (no pun intended) 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.



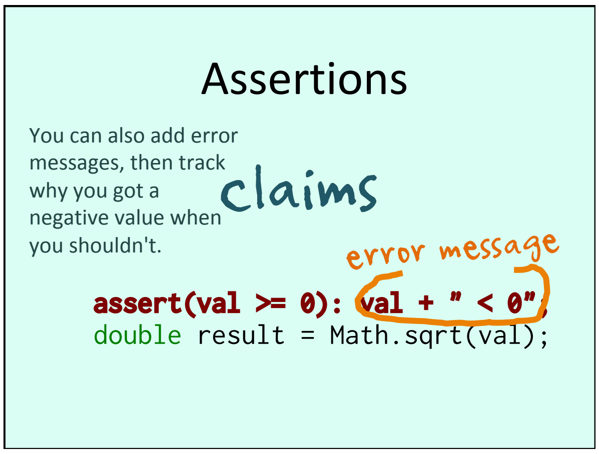
Assertions

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

claims

AssertionError

```
assert(val >= 0);
double result = Math.sqrt(val);
```



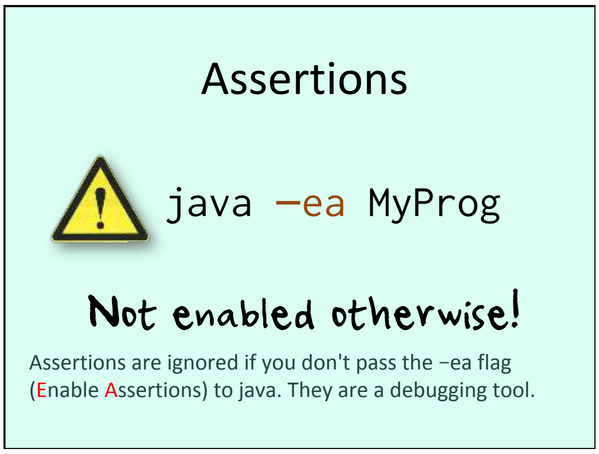
Assertions

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

claims

error message

```
assert(val >= 0): val + " < 0";
double result = Math.sqrt(val);
```



Assertions

```
java -ea MyProg
```

Not enabled otherwise!

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

RECURSION

Book Chapter 18

We are now going to move to a completely unrelated topic which is more about algorithms. It's not as vital when you code in Java as with some other languages. Nevertheless it's very important to know and very powerful. Before we talk about recursion, let's talk about sorts.

Sorts

This may not be your vision of computer science, but sorting is something that is and always has been a hot research topic. 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 incredibly important.

input: array of *n* values in random order

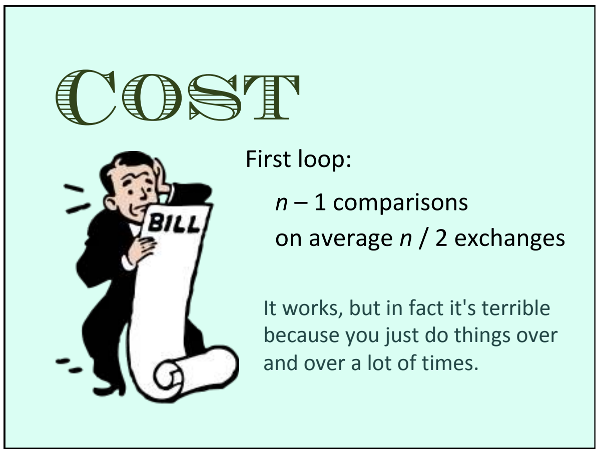
output: array of *n* sorted values

What we get initially and what we want is easy to define. Now how do we get from one to the other?

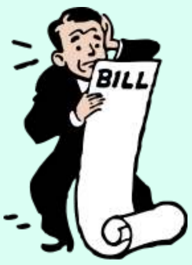
You may have heard about the "bubble sort": you start at the left end of the array and move to the right, exchanging values that aren't in the correct order. At the end of the first pass, the biggest value is correctly placed. Then you repeat, to place the 2^{nd} biggest, and so forth.

55	50	78	81	6	2	95	28	93	46

50	55	78	6	2	81	28	93	46	95



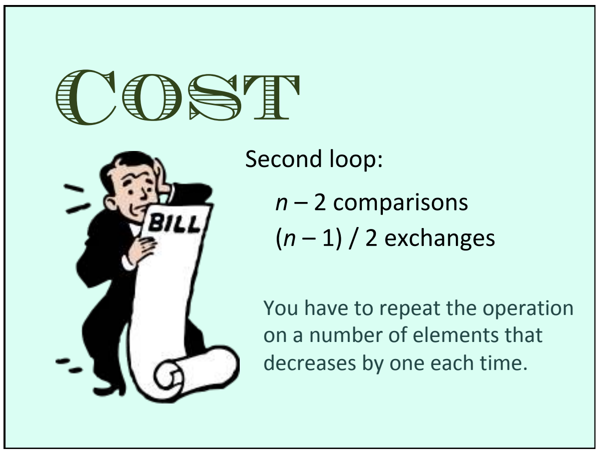
COST



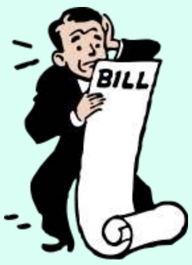
First loop:

- $n - 1$ comparisons
- on average $n / 2$ exchanges

It works, but in fact it's terrible because you just do things over and over a lot of times.



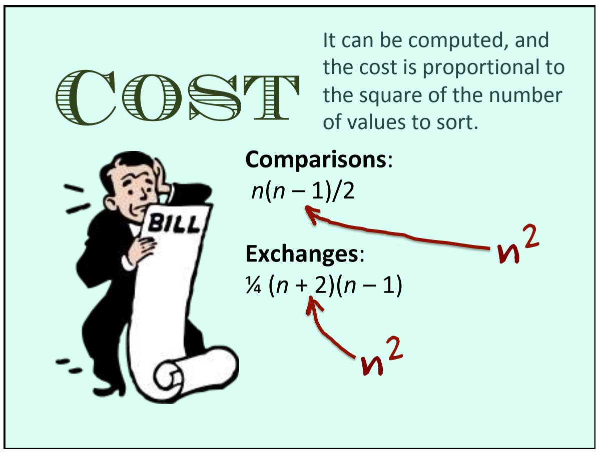
COST



Second loop:

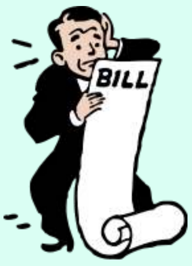
- $n - 2$ comparisons
- $(n - 1) / 2$ exchanges

You have to repeat the operation on a number of elements that decreases by one each time.



COST

It can be computed, and the cost is proportional to the square of the number of values to sort.

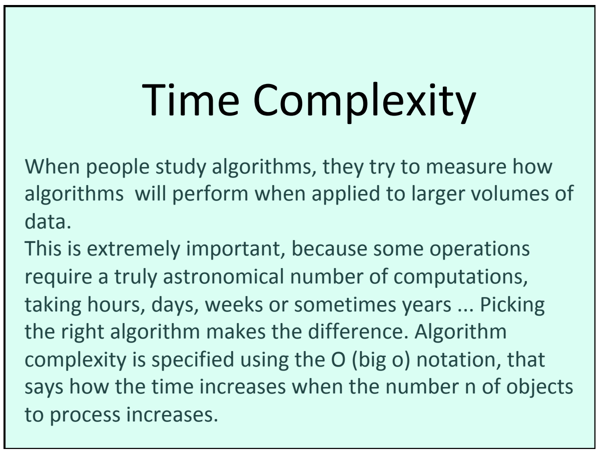


Comparisons:

$n(n - 1)/2$ $\leftarrow n^2$

Exchanges:

$\frac{1}{4}\,(n + 2)(n - 1)$ $\leftarrow n^2$



Time Complexity

When people study algorithms, they try to measure how algorithms will perform when applied to larger volumes of data.
This is extremely important, because some operations require a truly astronomical number of computations, taking hours, days, weeks or sometimes years ... Picking the right algorithm makes the difference. Algorithm complexity is specified using the O (big o) notation, that says how the time increases when the number n of objects to process increases.

$O(1)$

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.

$O(\log n)$

Means that the time evolves as the log. So, operating on 1000 more objects will multiply the time by 3 only (the log isn't necessarily decimal, it's the order of magnitude that counts), which is pretty good. Some search algorithms achieve this.

$O(n)$

Means that the time is directly proportional to the number of items processed. This is what you get when you scan an unordered array, looking for a value: if the size of the array doubles, it will take twice the time on average.

$O(n \log n)$

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 the bubble sort does. You definitely don't want to use that on large numbers of values.

Bubble Sort

Selection Sort

N^2

The "selection 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 the bubble sort, but is also an $O(n^2)$ sort. Yek.

$N_x\log(N)$

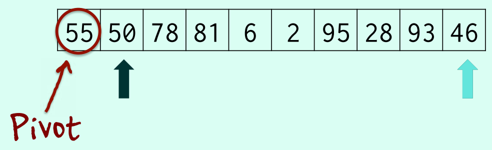
I have told it to you, the best sorting algorithms are O(nlogn) algorithms. What does it mean compared to a $O(n^2)$ algorithm?

N	N^2	$N_x\log(N)$
10	100	10
100	10000	200
1000	1000000	3000

Definitely more desirable. It increases faster than N, but not madly faster than N.

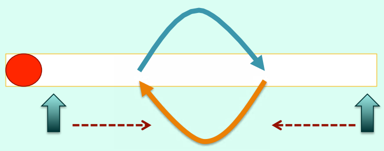
Quick Sort

The first really fast sorting alogrithm was invented by a young British mathematician, Antony Hoare (now Sir Antony Hoare) in the early 1960s when he was in Moscow.

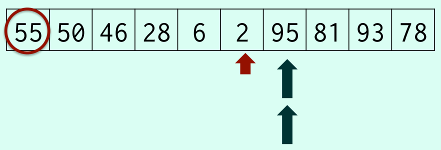


There are several brilliant ideas in the algorithm. One of them is, instead of successively looking for smallest (or greatest) values, to take arbitrarily one value called pivot and find its final location.

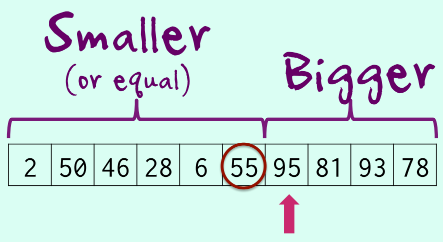
QuickSort



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.



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 red 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.

The **BIG** idea

N^2 Sorting twice the number of values is 4 times as costly

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.

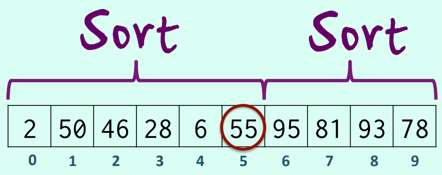
```java
static int placePivot(int[] arr, int first, int last) {
    int pivot;
    int tmp;
    int up = first + 1;
    int down = last;

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

Here is the Java code to find where the pivot goes: first we move two indices 'up' and 'down' until they find a value respectively greater and smaller than the pivot ...

```java
       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 met the down pointer
    if (pivot < arr[up]) {
       // Place pivot at up - 1
       up--;
    }
    arr[first] = arr[up];
    arr[up] = pivot;
    return up;
}
```

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

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" ...

Mathematical Parenthesis

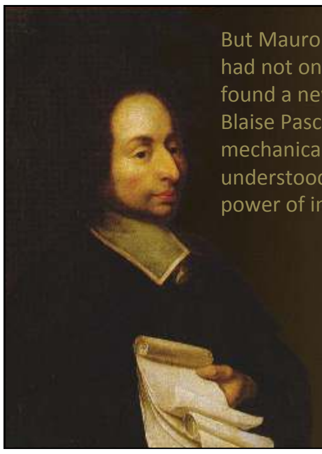
Thankfully, maths come to the rescue; not maths themselves, but a mathematical method which is very closely related to what we'll do.

MATHEMATICAL INDUCTION

This closely related mathematical method is induction, a very clever way of proving theorems.

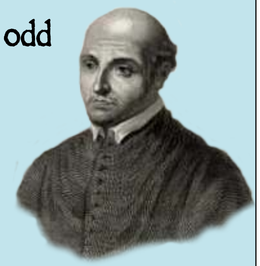
D. FRANCISCI
MAVROLYCI
ABBATIS MESSANENSIS,
Opuſcula Mathematica;
Nunc primum in lucem edita, cum rerum omnium notatu dignarum,
INDICE LOCVPLETISSIMO.
TABELLA HVIC PROXIME CONTIGVA, eorum Catalogus eſt.
CVM PRIVILEGIO.
Venetijs, Apud Franciſcum Franciſcium Senenſem.
M D L X X V.

The first one to have used it (in 1575) is Francesco Maurolico, an Italian professor at the University of Messina (opposite Sicily).



But Maurolico didn't quite realize he had not only proved a theorem, but found a new way of reasoning. It's Blaise Pascal (of, among other things, mechanical computer fame) who fully understood in the next century the power of induction.

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



This is what Maurolico proved, and to do it he used a two step method:

- Obvious for 1
- If it's true for N, it's true for N+1

Therefore it's true for any integer value.

Obvious for 0 and 1.

Suppose true for n.

$$1 + 3 + \ldots + (2n - 1) = n^2$$

The sum of the (n+1) first odd integers is

$$\underbrace{1 + 3 + \ldots + (2n - 1)}_{n^2} + (2n + 1)$$

$$n^2 + (2n + 1) = (n + 1)^2$$

It means "this is what had to be proved"; often shortened to QED in modern English (it's Latin) $\rightarrow$ Quod Erat Demonstrandum

Mathematical induction is based on these two elements:

Link between *n* and *n + 1*

Trivial case

Related (although not identical) thought in programming:

RECURSION

Recursion is also based on a link between n and n+ 1 and a trivial case, but it works in reverse order.

Mathematical Induction

Link between *n* and *n + 1*

⬆

Trivial case

Mathematical induction goes from the trivial case towards infinity.

Recursion

Link between *n + 1* and *n*

⬇

Trivial case

Recursion, which we'll use for the Quick-Sort, works by identifying a trivial case, then by 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

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

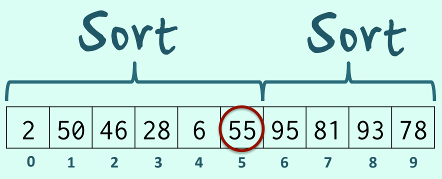
Recursion is characterized by functions that contain calls to themselves, with different parameters corresponding to a smaller problem.

~~**re·cur·sive** *adjective* \ri-ˈkər-siv\ *See* recursive~~

Not the way it works

Of course at one 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.

With a recursive function you MUST first identify trivial cases.



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

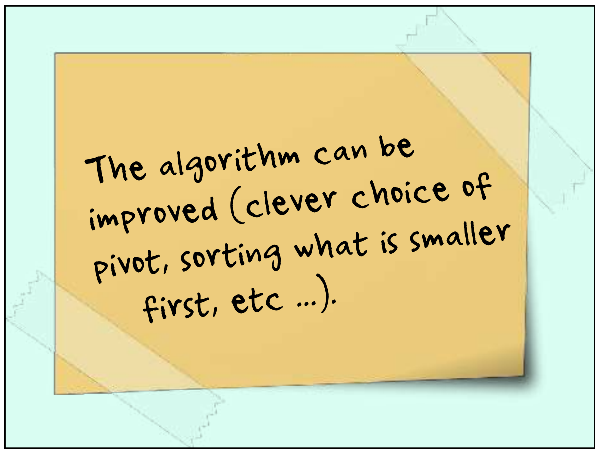
Trivial case ? Cases of 0 or 1 value to sort are kind of easy ...

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

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

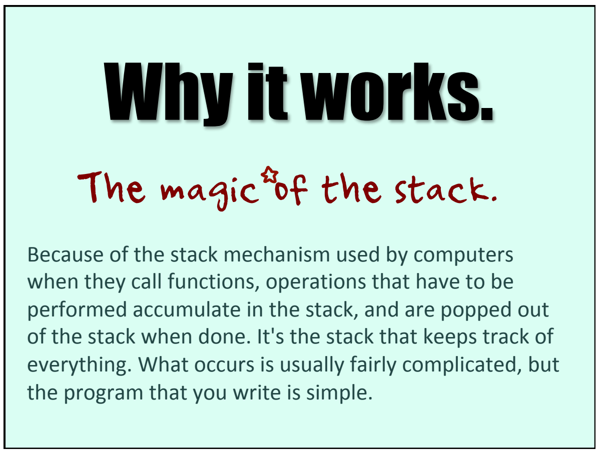
If there is 0 or 1 value in the array, nothing to do. If there are two values in the array we just check that they are in order and swap them if not.

```java
      switch (last - first) {
        case 1:
             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);
             quickSort(arr, pivotPos+1, last);
             break;
      }
    }
}
```

In the general case, we place the pivot, then call the function again for sorting the two subsets. And THAT'S ALL!



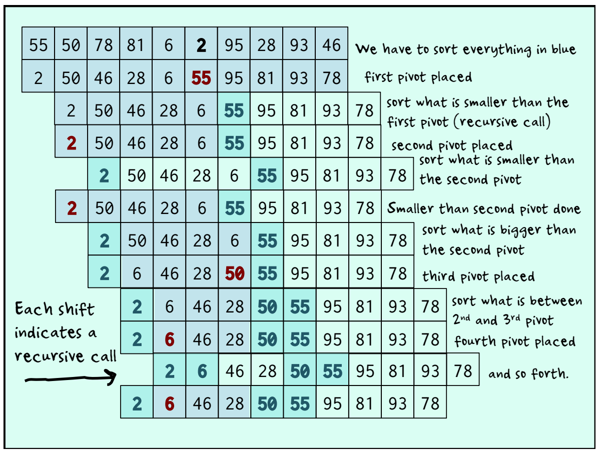
The algorithm can be improved (clever choice of pivot, sorting what is smaller first, etc ...).



Why it works.

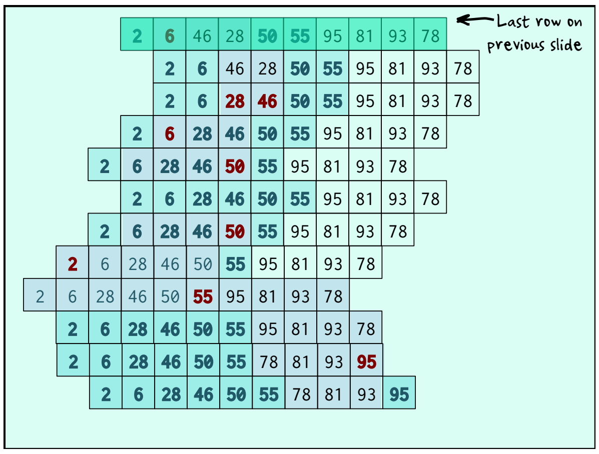
The magic of the stack.

Because of the stack mechanism used by computers when they call functions, operations that have to be performed accumulate in the stack, and are popped out of the stack when done. It's the stack that keeps track of everything. What occurs is usually fairly complicated, but the program that you write is simple.

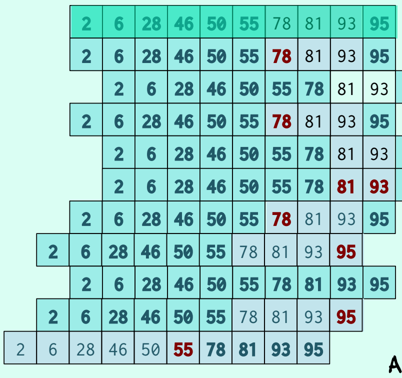


55	50	78	81	6	**2**	95	28	93	46					We have to sort everything in blue
2	50	46	28	6	**55**	95	81	93	78					first pivot placed
	2	50	46	28	6	**55**	95	81	93	78				sort what is smaller than the first pivot (recursive call)
	2	50	46	28	6	**55**	95	81	93	78				second pivot placed
		2	50	46	28	6	**55**	95	81	93	78			sort what is smaller than the second pivot
	2	50	46	28	6	**55**	95	81	93	78				Smaller than second pivot done
		2	50	46	28	6	**55**	95	81	93	78			sort what is bigger than the second pivot
		2	6	46	28	**50**	**55**	95	81	93	78			third pivot placed
			2	6	46	28	**50**	**55**	95	81	93	78		sort what is between 2nd and 3rd pivot
			2	**6**	46	28	**50**	**55**	95	81	93	78		fourth pivot placed
				2	**6**	46	28	**50**	**55**	95	81	93	78	and so forth.
			2	**6**	46	28	**50**	**55**	95	81	93	78		

Each shift indicates a recursive call $\longrightarrow$



			2	**6**	46	28	**50**	**55**	95	81	93	78		$\longleftarrow$ Last row on previous slide
				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**	95	81	93	78				
	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)

Execution is horribly complicated ...

...but writing is easy.

DON'T use recursion where loops are easy to write.

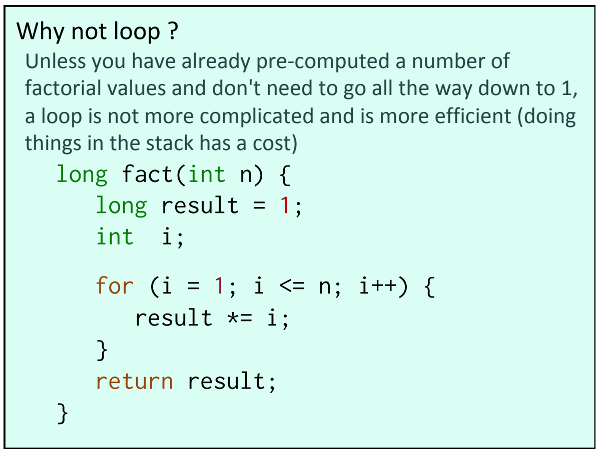
Even if recursion can be an easy, and elegant, way of solving hairy problems, the big benefit of recursion is when operations to perform multiply out of control - the quick sort is a good example because everytime you want to sort a set, you end up with two smaller sets to sort.

If you know the sorcerer's apprentice in Disney's "Fantasia" ...

Traditional (stupid IMHO) recursion example:

As a factorial can be defined as
n! = n x (n-1)!
it is often used as an example for teaching recursion.

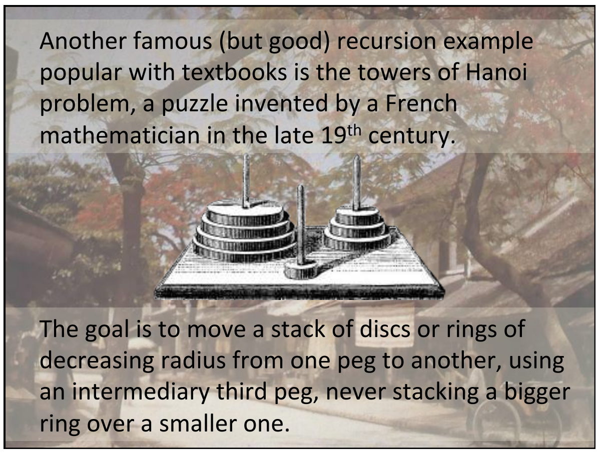
```
long fact(int n) {
    if (n == 0) {
        return (long)1;     Trivial case
    } else {
        return n * fact(n - 1);Recursion
    }
}
```



Why not loop ?

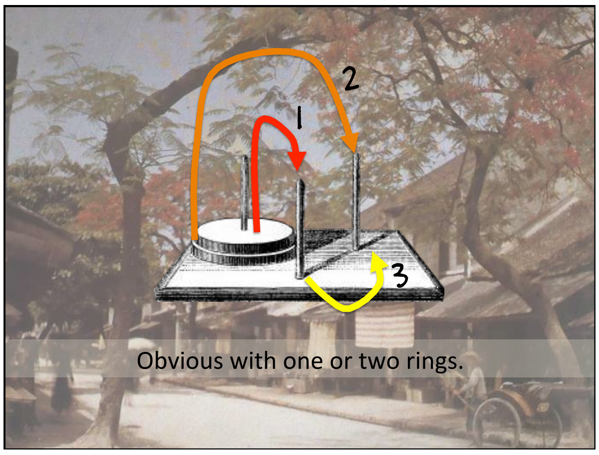
Unless you have already pre-computed a number of factorial values and don't need to go all the way down to 1, a loop is not more complicated and is more efficient (doing things in the stack has a cost)

```
long fact(int n) {
   long result = 1;
   int  i;

   for (i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

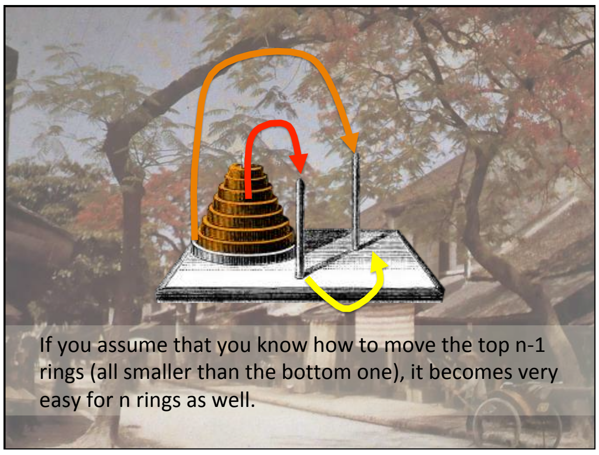


Another famous (but good) recursion example popular with textbooks is the towers of Hanoi problem, a puzzle invented by a French mathematician in the late 19th century.

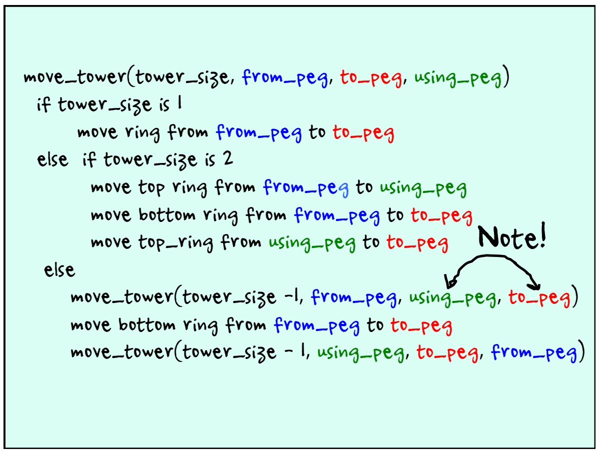
The goal is to move a stack of discs or rings of decreasing radius from one peg to another, using an intermediary third peg, never stacking a bigger ring over a smaller one.



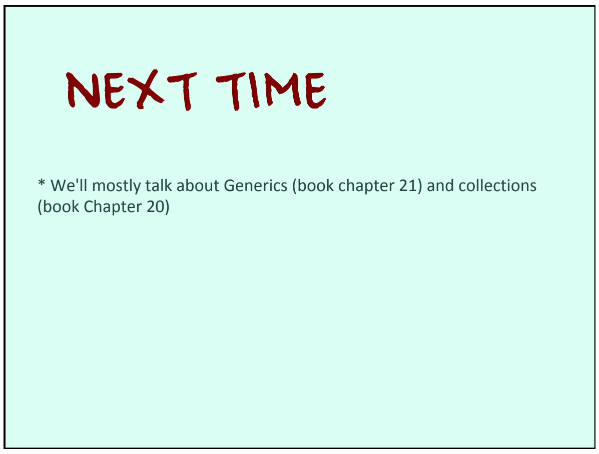
Obvious with one or two rings.



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.



```
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      Note!
  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)
```



NEXT TIME

* We'll mostly talk about Generics (book chapter 21) and collections (book Chapter 20)