

CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn
Liu Zijian liuzijian47@163.com
Li Guansong intofor@163.com

As we have seen last time, you can either catch exceptions, or pass them back to the caller, either by declaring that the method throws the exception if it's a checked exception, or by doing nothing if it's an unchecked exception.

The practical problem, when a program starts to generate exceptions too often, is usually to find exactly where they first occur and, perhaps, try to fix them there.

As we have seen last time, Errors and Exceptions all extend the Throwable class that implements some very interesting constructors and methods.

Throwable

```
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

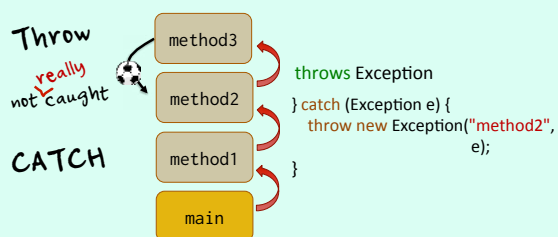
What can we do with
"unknown unknowns"?

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.



```

} 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



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.

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

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

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

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



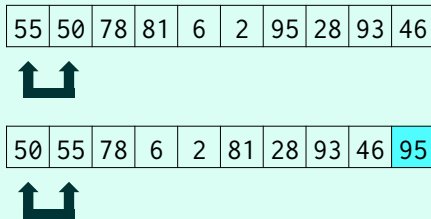
This may not be your vision of computer science, but sorting is something that is and always has been a hot research topic. Computers mostly help us manage data; most programs, before they compute anything, have to retrieve data. And for retrieving data easily, data must be sorted, or kept in order. Sorting efficiently is incredibly important.

input: array of n values in random order

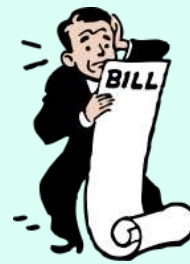
output: array of n sorted values

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 2nd biggest, and so forth.



COST

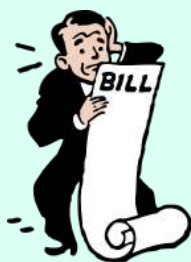


First loop:

$n - 1$ 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:

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

Exchanges:

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

n^2

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 n^{th} 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 1,000,000. 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.

 $N \times \log(N)$

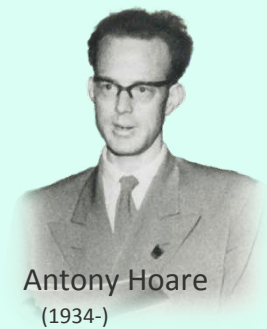
I have told it to you, the best sorting algorithms are $O(n \log n)$ algorithms. What does it mean compared to a $O(n^2)$ algorithm?

N	N^2	$N \times \log(N)$
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

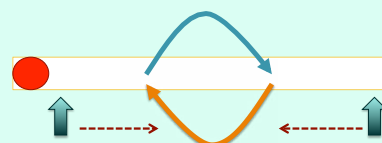
The first really fast sorting algorithm was invented by a young British mathematician, Antony Hoare (now Sir Antony Hoare) in the early 1960s when he was in Moscow.



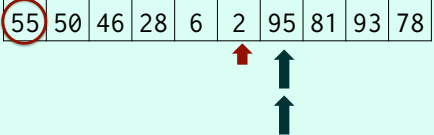
Pivot

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.

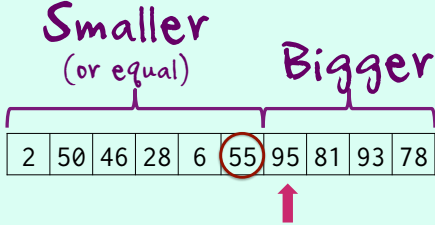
QuickSort



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.

```
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--;
        }
        tmp = arr[up];
        arr[up] = arr[down];
        arr[down] = tmp;
    }
    tmp = arr[first];
    arr[first] = arr[up];
    arr[up] = tmp;
    return up;
}
```

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 ...

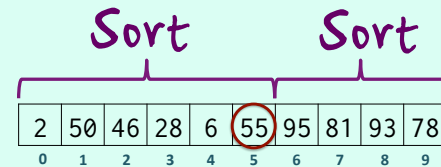
```

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

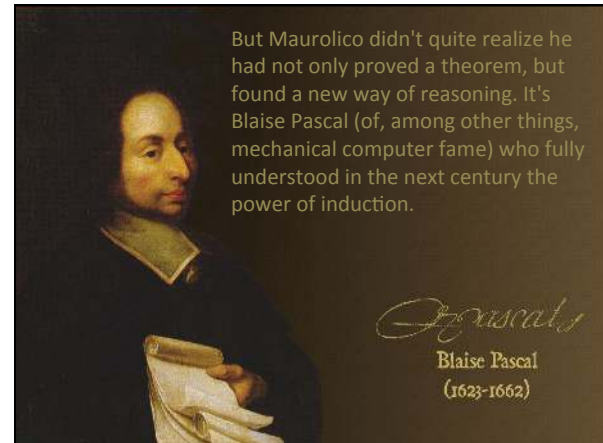
"Remember" ...

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.




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.

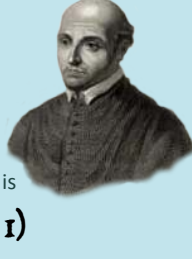
Suppose true for n .

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

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

$$1 + 3 + \dots + (2n - 1) + (2n + 1)$$

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

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


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).

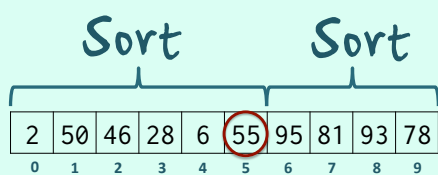
Recursion is characterized by functions that contain calls to themselves, with different parameters corresponding to a smaller problem.

~~re·cur·sive adjective \rɪˈkɔːsɪv/ 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.



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

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

```
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.

```

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!

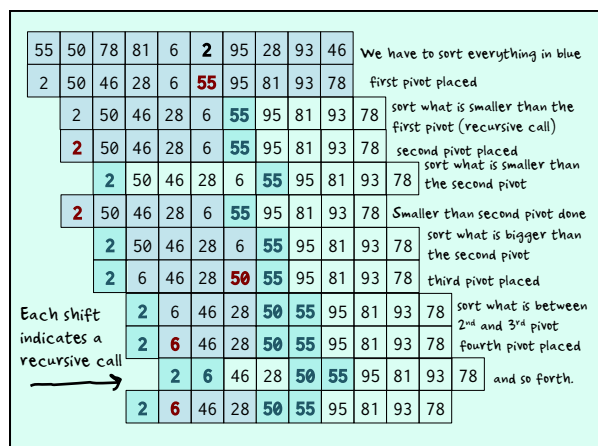
```

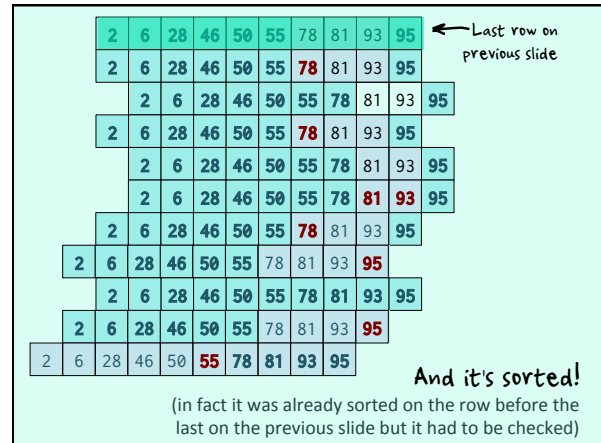
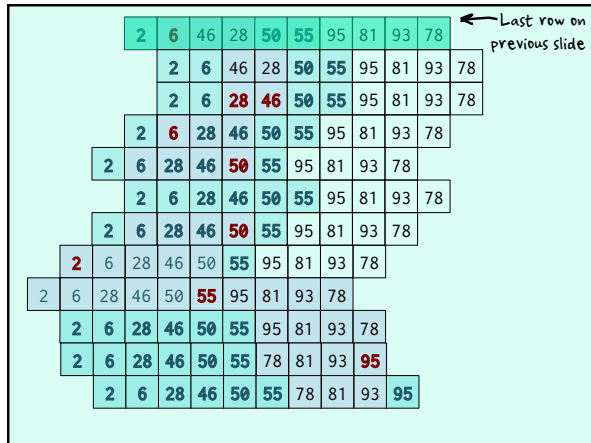
The algorithm can be improved (clever choice of pivot, sorting what is smaller first, etc ...).

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.





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

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

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 ?

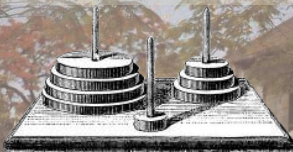
Unless you have already pre-computed a number of factorial values and don't need to go all the way down to 1, a loop is not more complicated and is more efficient (doing things in the stack has a cost)

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

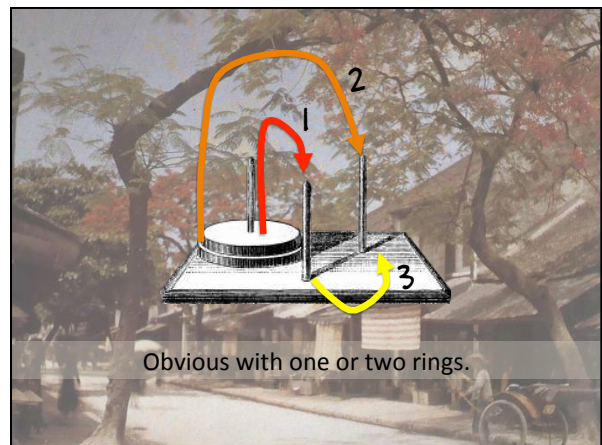
    for (i = 1; i <= n; i++) {
        result *= i;
    }

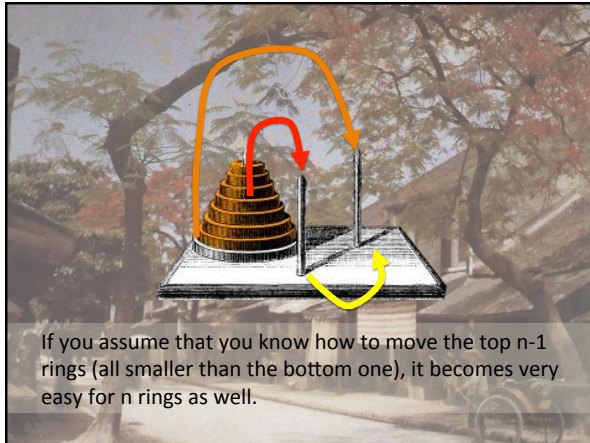
    return result;
}
```

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.





```

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)
  
```

Note!

Recursion is very important with complex collections of objects

We have seen it with the quick-sort: inside an array you can find smaller sub-arrays. Inside a set you find subsets. And when you subdivide everything enough, you find empty subsets, or subsets that contain a single element. It's a fantastic ground to apply recursion.

Java Generics

Book Chapter 21

But before we talk about collections of objects, we have to talk about "Generics" (an idea very similar to "Templates" in C++, although implementation is quite different), which are very important for reusing code in Java. You may have already had some exposure to Generics through ArrayLists.

Most languages that have the ambition of being used in critical applications insist on "strong typing". It means that the compiler will NOT let you mix variables of different types, unless they are known to be compatible (such as int and float). It protects against unwanted effects (note that many scripting languages take the opposite approach and guess the type of variables from how you are using them).

Java = strong typing
SAFE
but ...

```
String[] months = {"Jan", "Feb", "Mar", "Apr",
                  "May", "Jun", "Jul", "Aug",
                  "Sep", "Oct", "Nov", "Dec"};
int[] sales = {120, 98, 75, 110, 150,
               180, 170, 174};
```

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
120 98 75 110 150 180 170 174
```

The problem with strong typing is that for instance if you want to write a methods that displays on a line elements from an array, the same method cannot handle an array of Strings and an array of ints. It's one or the other, exclusively.

```
public static void displayArr(String[] arr) {
    int n = arr.length;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            System.out.print("\t");
        }
        System.out.print(arr[i]);
    }
    System.out.println("");
}
```

If this is the function that displays an array of Strings ...


```
public static void displayArr(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            System.out.print("\t");
        }
        System.out.print(arr[i]);
    }
    System.out.println("");
}
```

Overloading

... you must overload it to display an array of integers. Note that, apart from the parameter type, the code is strictly identical.

Overloading allows you to give the same name to several functions. By looking at parameters, Java will know exactly which one to call.



Overloading

Different versions of a function

- Same name
- Same return type
- Different parameters

Flickr: Tim McCune

It's the linker of the class loader that will match your code to methods with the same name that are available.

What defines the "signature" of a function?

Number and types of parameters

subString(String s, int start)

subString(String s, int start, int len)

isGreater(double, double)

isGreater(String, String)

isGreater(Date, Date)

```
public class Overloading {
    static int function(int n) {
        System.out.println("This is function(int)");
        return 0;
    }
    static int function(double x) {
        System.out.println("This is function(double)");
        return 0;
    }
    public static void main(String[] args) {
        int n = 1;
        double val = 0;
        float f = 0;
        n = function(n);
        n = function(val);
        n = function(f);
    }
}
```

In this example, when we call the function with a float, it's automatically "upgraded" to double and the function for doubles is called (the same function would be called for the int if there were no special function for integers)

If overloading helps keep the code safe, it's a waste of time to write identical code several times (even if we copy and paste). Worse, if we want to change the code, for instance to separate array elements by semicolons (;) instead of tabs when we print them, we must modify every method.

Waste of time.

Change: must be repeated in all methods.

Code that works with Strings and integers?

BEWARE: in Java, generics ONLY WORK with objects (references). Base variables, such as int, float, char, boolean variables ARE NOT objects. However, all base types have a "shadow" corresponding class (Integer, Float, Character, Boolean ...). For using generics, we must use these classes, which convert automatically to and from base data types (operations known as "boxing" and "unboxing")

IMPORTANT

What we'll see only works with OBJECTS

String

int → Integer

```
String[] months = {"Jan", "Feb", "Mar", "Apr",
                  "May", "Jun", "Jul", "Aug",
                  "Sep", "Oct", "Nov", "Dec"};
Integer[] sales = {120, 98, 75, 110, 150,
                  180, 170, 174};
```

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
120 98 75 110 150 180 170 174
```

If we have an array of Strings (which are objects) and an array of Integers (that are objects too), then we can create a single method that works for both, without any explicit overloading.

"Generics" comes from a Latin word that means "family" or "kind". It can be seen as automatic overloading. Once again (it's worth repeating) it only works with references in Java.

GENERICS

Computer-aided Overloading

Only works with references

```
public static <T> void displayArray(T[] arr) {
    int n = arr.length;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            System.out.print("\t");
        }
        System.out.print(arr[i]);
    }
    System.out.println("");
}
```

```
displayArray(months);
displayArray(sales);
```

Before the return type of the method, you specify one (or several) symbols between angular brackets that represent Classes.

You need to specify the symbol before the return type because the return type of the generic method could perfectly be defined as T. If you need several generic classes (for instance one for the return type and one for the parameter, or because you want to pass parameters from two different classes), you separate them with commas.

<T,U>

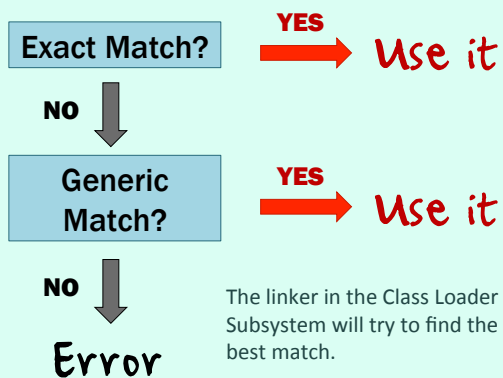
Generic methods can be overloaded

```
public static <T> void displayArray(T[] arr)

public static <T> void displayArray(T[] arr,
                                    String sep)

public static void displayArray(Date[] arr,
                                String format)
```

Like any method, generic methods can be overloaded, either by another generic method, or for instance to handle a very special case.



Functions can be generic.

Classes can also be generic.

Important for collections!

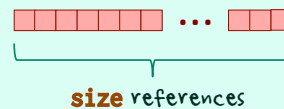
This idea of turning a class into a kind of parameter can also be applied to classes; this is important for collections, which are "container classes". For instance, an ArrayList is a class that can hold (contain) objects of any class.

Collections

Let's move to collections by reviewing the different ways we have to group objects together.

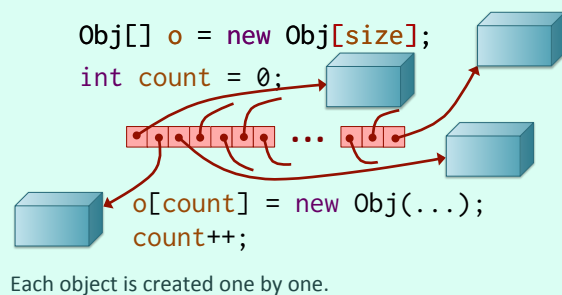
Arrays of Objects

```
Obj[] o = new Obj[size];
int count = 0;
```



When you create an array of objects, each element isn't an object (contrary to arrays of base types). It's just a place to store a reference.

Arrays of Objects



Arrays of Objects

And then?

```
Obj[] o2 = new Obj[bigger];
```

o = o2;
o2 = null;

If you exceed the size of the array, you can create a bigger one and copy the references there.

NEXT TIME

* We'll mostly talk about collections (book Chapter 22)