G51OOP

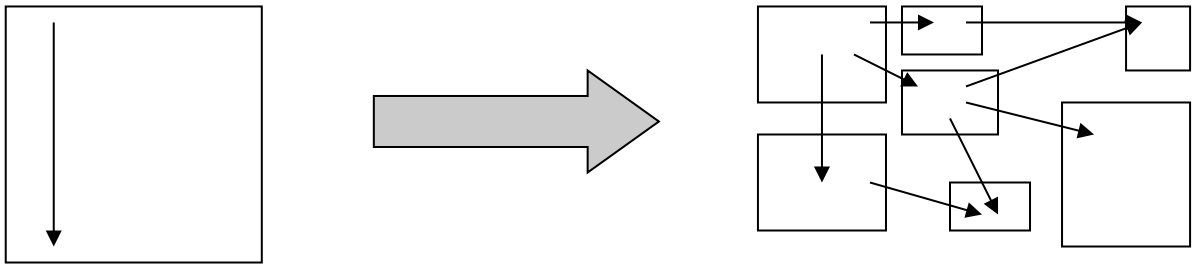
2. Methods

Object Oriented Programming

Colin Higgins

Motivation: the need for modularity.

- Methods enable us to break a program down into a number of smaller units. This is taken one step further later in the course with the idea of classes and objects. There are several different reasons why programs should be broken down in these ways.



(i) Most problems naturally break down into sub-problems.

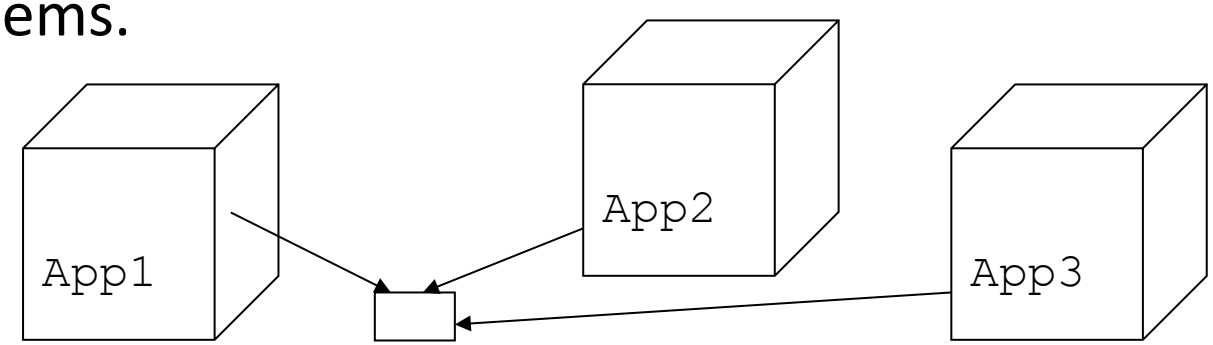
(ii) Large problems must be implemented by a team, not an individual.

(iii) Reusability of the parts, is highly appreciated

  - For the program itself
  - For other programs

Motivation: the need for modularity.

- Units which solve frequently occurring sub-problems can be reused in appropriate places in a variety of larger problems.



(i) They may be used in order to save effort in re-solving that particular problem, and in re-programming the solution.

(ii) They may be used in order to produce better quality systems, on the assumption that the solution being re-used was tested to a high quality standard when it was originally written.

The syntax of a method

- A method consists of

```
[modifiers] <returned type> <identifier> (
[<param type> <param ident>],
[<param type> <param ident>],
....
) {
   // fields and statements
   // a return statement
}  // usually an end comment here
```

- The returned type may be any ordinary Java type such as "int", "float" and so on, or "void" if no value is being returned to the main program.
- Later we will see how to return objects.

A simple example of a method

```
public class MethodExample {
   // First the declaration and definition
   // of two methods – “static” for now
   static void dothis() {
      System.out.println("Hello");
   } // end of dothis

   static void dothat() {
      System.out.println(“Hello Again!");
   } // end of dothat
```

A simple example of a method

```
// Now the program calls them
  public static void main(String argv[]) {
      // These are the calls
      dothis();
         ....;
      dothat();
   } // end of main
} // end class MethodExample
```

Examples of methods

```
public static double myFunction (int intParameter,
                    float floatParameter ){
    double myLocalVariable;
    myLocalVariable = intParameter * floatParameter;
    return myLocalVariable;
}

public static int third_power (int number) {
     int cube;
     cube = number * number * number;
     return cube;
}  // method third_power
```

Examples of method declarations

- A method to compute the power of 2 of a given value.

```
// identifier “power2” - takes one double parameter
// gives double result

double power2(double value) {
    return value*value;
}
```

- A method to detect the larger of two integer values.

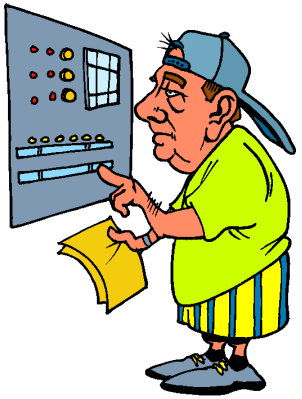
```
// identifier "max“ takes two integer parameters,
// and gives an integer result
int max(int arg1, int arg2) {
   if (arg1 > arg2)
     return arg1;
   else
     return arg2;
}
```

Examples of method declarations

- A method to print an error message involving an integer value.

```
// identifier "error"
// takes integer parameter, gives no returned result
public static void error(int errorNumber) {
   switch (errorNumber) {
       case 1:
              System.out.println(“Bad Command”);
              break;
       case 2:
              System.out.println(“Wrong parameters”);
              break;
       ...
    }
}
```

Method calls



- The above methods would be called from elsewhere (perhaps from `main()`, perhaps from another method using similar statements.

- To make a method call (to a method in the current class) you simply name the method in an appropriate statement and give actual parameters (which will "take the place" of the formal parameters).

Example 1: Invoking a method

- Square root would take a "double" parameter and return a double result.

```
double y = sqrt ( 2.0 );
....
if ( sqrt ( x * 5 ) > 2.0 ) {
    ....
}
....
System.out.println("Sqrt " + sqrt( 2.0 ) );
```

- We would normally expect to use the returned result as a value.

Example 2 : Invoking a method

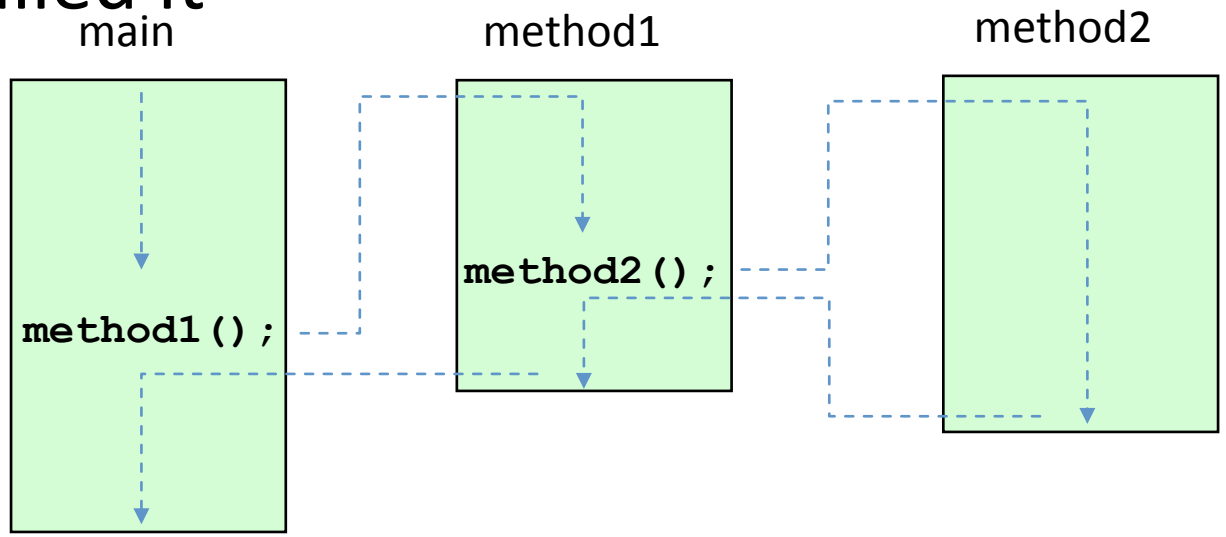
```
int biggest, mark, nonNegative,first,second;
System.out.print("Please type in two numbers: ");
System.out.flush();
first = UserInput.readInt();
second = UserInput.readInt();
biggest = max( first, second );
mark = max( first, 40 );
System.out.println("Larger value is " + max ( first, second));
nonNegative = max( first, 0 );
```

Example 3. Invoking a method

```
// Reporting an error
if ( n < 0 ) {
   error ( 5 ); // no result
} else if ( n > 100 ) {
   error ( 6 ); // no result
}
```

Method Flow of Control

- The `main` method is invoked by the system when you submit the bytecode to the interpreter
- Each method call returns to the place that called it



The exit method

- Convention with the "exit" function is that :

```
exit( 0 ); // indicates normal exit
exit( 1 ); // indicates error exit
exit( 2 ); // indicates error exit
```

- The UNIX/Windows/etc shell can use the value returned by a terminating program to determined whether it terminated successfully or not.

Returned types

- The compiler needs to know the type of the value to be returned by the methods so that it can be treated correctly in the call of the method.If no value is to be returned (like in "error" or "dothis" above) then the return type is given as "void".

- The particular value to be returned is specified by the line `return <value>;`
- With a "void" method you have the option of using "`return;`"

Parameter types

- The parameter type sequence in the declaration and the definition must agree. The compiler will make the necessary type conversions to the parameters and to the delivered result in a call of the method.
- Again, the compiler will perform type conversions if the method declaration specifies a "float" parameter, and if the call includes an actual integer parameter.

- Remember the parameters in the method definition are called the formal parameters. The parameter values supplied in calls of the method are called actual parameters.

Method overloading

- You can use the same method identifier for more than one method, providing that the compiler can determine the particular one to be chosen by the types (and number) of its parameters.

```
int square ( int i ) {
   // int squared is an int
   return i * i;
}

double square ( double x ) {
   // double squared is a double
   return x * x;
}

double square ( double x, double height ) {
   // return the volume of a square prism
   return x * x * height;
}
```

Method overloading

- The calls might be :

```
i = square( 3 );
double z;

if ( square( z ) > 3.14159 ) {
   ....
}

vol = square( z, 1.27 );
```

Local variables

- We can declare local variables at the start of our method code if we require additional variables for computations within the method. The declarations appear after the opening curly brace, just as they do in main().
- If a local identifier clashes with a class constant or variable, or with the name of another method, then that global constant or other method becomes inaccessible within this method. The local object will be referred to by the identifier.

```
float myFloat = 17.5F;

public static void main(String [] argv) {
    ...
    myFloat ... // global to class
}

static int fred( int i ) {
    float myFloat; // local variable
    myFloat = ...; // local variable
}
```

Parameters called by value

- When a method is called, the values of the (actual) parameters at the point where the method is called are calculated, and passed to the method definition for execution. Within the method, the parameters act like variables, initialised to the value of the corresponding actual parameter at the call, and their values can be changed by ordinary assignment.
- Thus if a method definition is :

```
int fred( int number ) {
    ....;
    number = number + p;
    ....;
} // end method fred
```

the identifier "number" can be considered as a local variable to the method. Changing its value inside the method as shown in the above example has no effect on the outside world.

A complete example

- The "Halberstam Function" of an integer value is defined as follows:
  - Given a positive integer value, generate the sequence
  - If it is even, half it.
  - Otherwise multiply by 3 and add 1.

- The Halberstam value of an integer value "n" is the number of times this operation needs to be repeated before the value 1 (one) is reached.
- It is guaranteed that you will eventually reach the value 1.
- If we start with the value 3, for example, the sequence goes 10, 5, 16, 8, 4, 2, 1; 7 steps.
- If we start with the value 7, the sequence goes 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1; 16 steps.
- A complete program with function and main program will look roughly as follows. We use an example with the Halberstam function.
- main prompts the use for a starting value and prints the result.

Halberstam Solution

```
// The main program
public static void main(String[] argv) {
    int number = 0, calcs = 0;

    System.out.print("Enter the number now please: ");
    number = G51OOPInput.readInt();

    while (number != 0) {
        calcs = halberstam(number);
        System.out.print("numb " + number);
        System.out.println(" result " + calcs);
        System.out.print("Enter the number now please: ");
        System.out.flush();
        number = G51OOPInput.readInt();
    }
}
```

```
// Halberstam function, by Eric Foxley and Colin Higgins
import java.lang.*;
public class Halberstam {
    static int halberstam(int number) {
        int calcs = 0;

        // Error exit
        if (number <= 0) {
            return -1;
        }

        // Loop counting how many times
        while (number > 1) {
            if (number % 2 != 0) { // number is odd
                number = number*3 + 1;
            } else { // number is even
                number /= 2;
            }
            calcs++;
        } // End of while loop

        return calcs;
    }
```

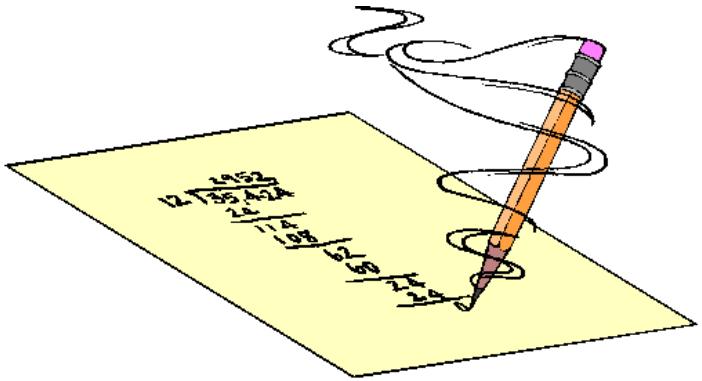
Halberstam Solution

Mathematical methods

- A number of standard mathematical methods are available. For details see the java.lang.Math class.

Basic numerical operations (Equivalent to C's <math.h>)

exponentials
logarithms
square roots
trigonometrics
etc. etc.



Examples:

`abs`
`pow, sqrt`
`log, exp`
`sin, cos, tan`
`asin, acos,`
`atan`
`floor, ceil, round`
`max, min`
`random`

Recursion

- "Recursion" occurs when you invoke a method within itself. Obviously you must be careful not to let the method to recurse indefinitely (infinitely), so a recursive method will usually have an "escape clause" such as "if `( n <= 1 ) ...` "

Factorial Example

- Consider a method which will produce the factorial of a positive integer value, where factorial(n) is defined as `n * (n-1) * (n-2) * … *  2 * 1.`
- We could do this with a for loop. However, this problem is more naturally solved recursively.
- We note that another way of defining factorial is

```
if (n == 1) factorial(n) = 1 otherwise
   factorial(n) = n * factorial(n – 1)
```

Solution

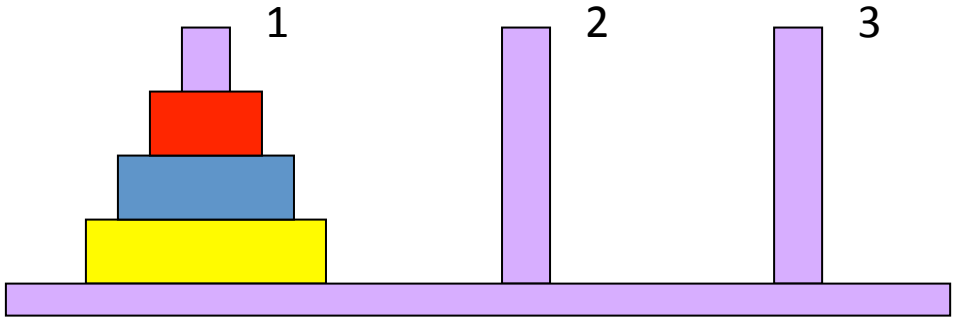
```
public static int factorial( int n ) {
    if ( n < 1 ) {
       return -1;      // report error
    } else if ( n == 1 ) {
       return 1;
    } else {
       return n * factorial( n - 1 );
    }
 }
```

The Towers of Hanoi problem.

- The problem is to get all the disks on one tower to another tower, moving one disk at a time and only being allowed to put a small disk on a larger one. There is a third tower available as a "temporary" resting place for disks.

Solution

- Number the towers 1 2 3.
- Start with 64 disks on 1. We need to move them to 3.
- Denote this task by movetower(64, 1, 3)
- We need to provide a list of correct moves to solve the problem. The insight is to think about the bottom disk on tower 1 rather than the top disk. movetower(64, 1, 3) can be achieved by
- movetower(63, 1, 2)
- move a disk from 1 to 3
- movetower(63, 2, 3)



The Tower of Hanoi problem.

Now movetower(63, 1, 2) can be achieved by :

movetower(62, 1, 3)

and

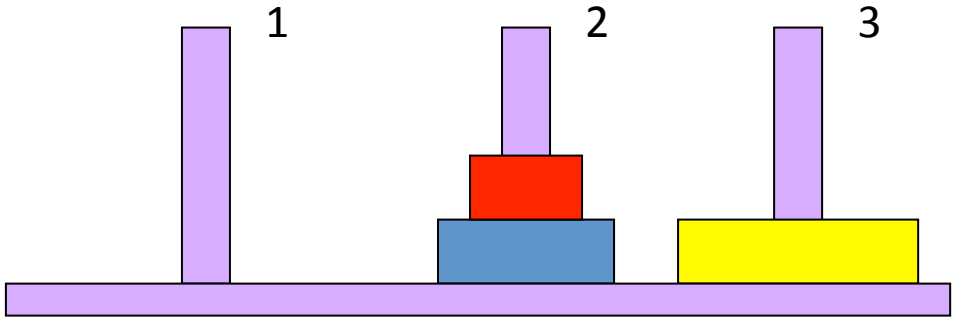
move a disk from 1 to

and

movetower(62, 3, 2)

To construct a general algorithm we need to specify which tower to use as a temporary tower. So movetower(n, a, b, c) means :

move n disks from a to b using c as the temporary tower.



The Tower of Hanoi problem.

movetower(n, a, b, c) can then be performed by the following three steps

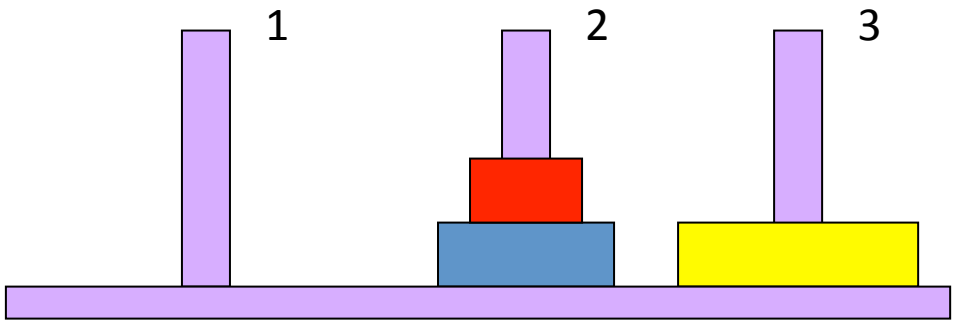
movetower(n-1, a, c, b)

move a disk from a to b

movetower(n-1, c, b, a)

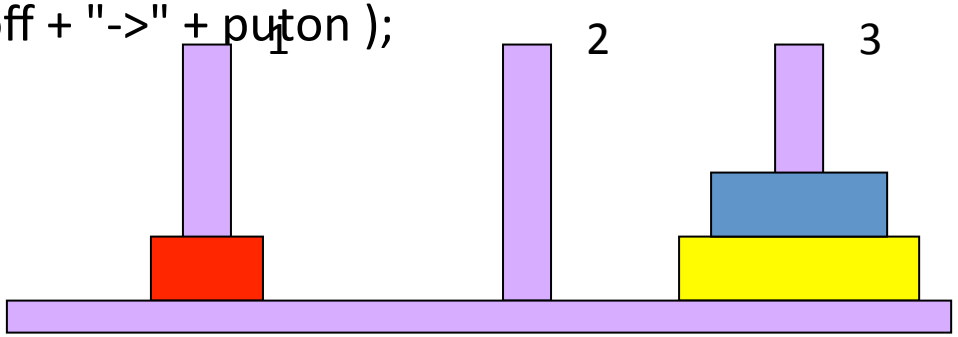
We need to cope with n <= 1 so add the rule :

if n <= 1 do nothing.



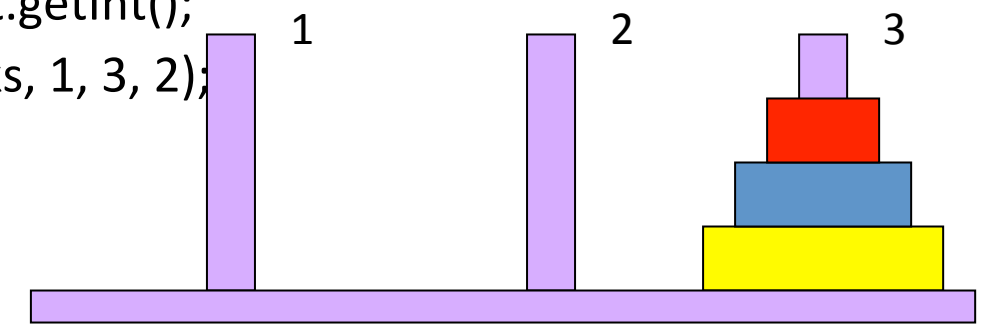
The Tower of Hanoi program

```
public class Hanoi {

    public static void movetower(int height, int fromT, int toT, int usingT) {
    if ( height > 0 ) {
            movetower( height –1, fromT, usingT, toT );
            moveDisk( fromT, toT );
            movetower( height –1, usingT, toT, fromT );
    }
}

    void moveDisk( int takeoff, int puton ) {
            System.out.println( takeoff + "->" + puton );
     }
```



The Tower of Hanoi program

```
    public static void main() {
        int numberOfDisks;
        System.out.print("how many disks?: ");
        System.out.flush();
        numberOfDisks = userInput.getInt();
        movetower( numberOfDisks, 1, 3, 2);
    } // end main
} end class Hanoi
```



- To learn more about this interesting solution look at : http://www.lhs.berkeley.edu/Java/Tower/towerhistory.html