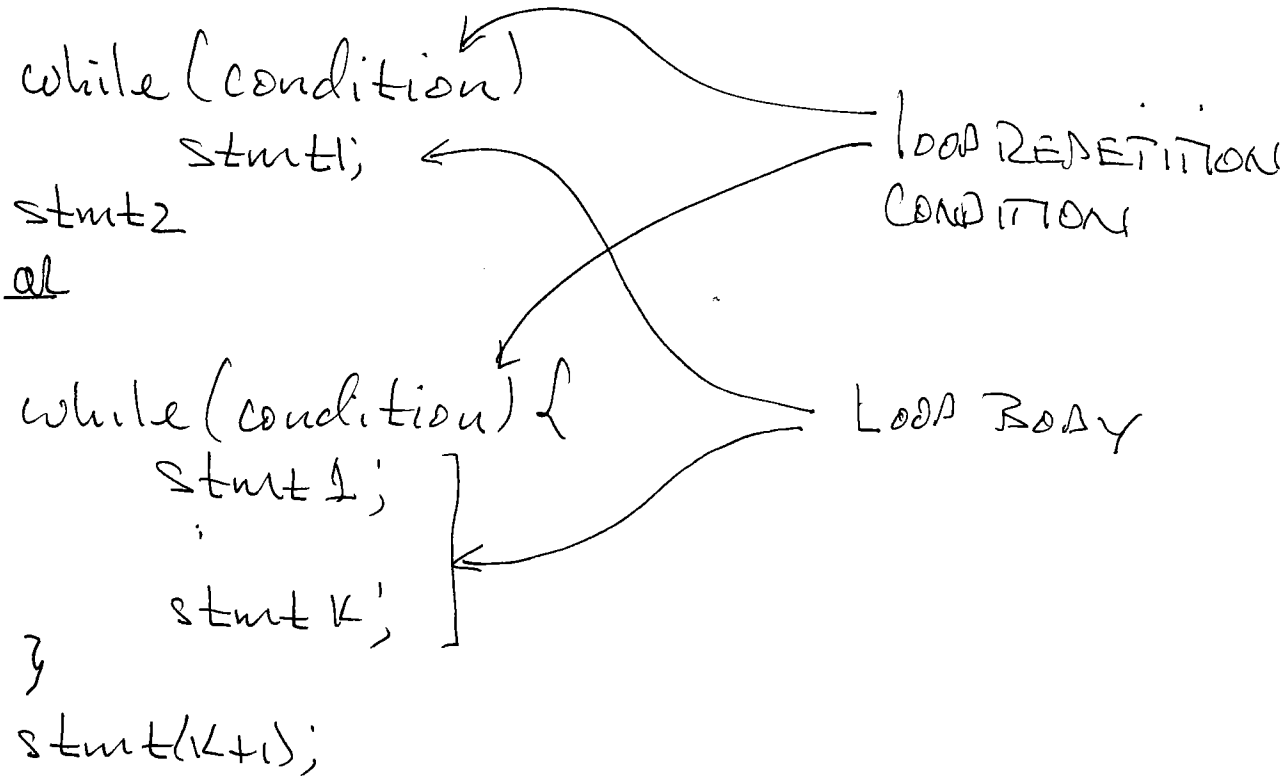
ITERATIVE (LOOPING) CONSTRUCTS

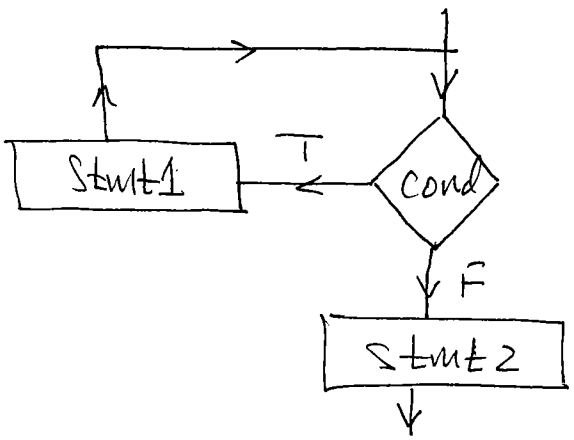
- while
- do-while
- for

FIRST THE while loop:



First condition is EVALUATED. IF TRUE stmt1 (or A COMPOUND stmt) is EXECUTED, THEN condition is TESTED AGAIN. AS LONG AS CONDITION is TRUE, stmt1 is EXECUTED.

THE FIRST TIME CONDITION IS FALSE, stmt1 IS SKIPPED AND EXECUTION RESUMES AT stmt2.



NOTICE THAT IT IS POSSIBLE FOR THE loop BODY TO EXECUTE ZERO TIMES, IF CONDITION IS INITIALLY FALSE

Ex.

```
int i, n = 7;

i = 1;
while (i <= n) {
    System.out.println(i * i);
    i += 1;
}
System.out.println("i = " + i);
```

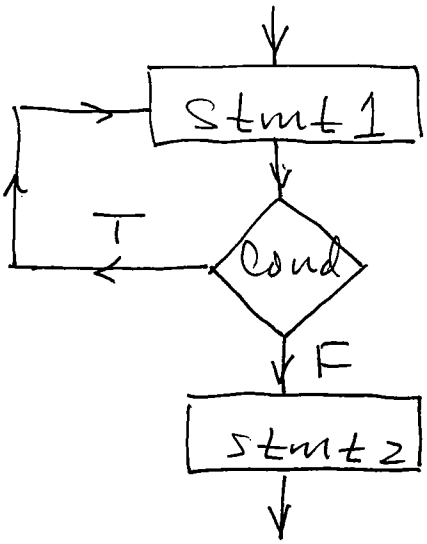
THE do-while loop IS VERY SIMILAR.

```
do
    stmt 1;          <-- loop body
while (condition);   <-- NOTE This semicolon
stmt 2;              <-- loop REPETITION Cond.
```

or

```
do {
    stmt 1
     ;
    stmt k;
} while (condition);
stmt (k+1);
```

Here the loop body is EXECUTED FIRST, then the loop REPETITION CONDITION is TESTED.



Ex

```
int i, n=7;

i=1;
do {
    System.out.println((int)Math.pow(i, 2));
    i++;
} while (i<=n);
System.out.println("i=" + i );
```

OBSERVE THAT IN THE do-while LOOP, THE LOOP BODY MUST EXECUTE AT LEAST ONCE.

WE OFTEN NEED TO WRITE A while LOOP OF THE FOLLOWING FORM:

```
i=1;
while(i<=n){
  //do something with i
  i=i+1;
}
```

INITIALIZE LOOP CONTROL VARIABLE

TEST LOOP REPETITION CONDITION

INCREMENT LOOP CONTROL VARIABLE

THIS IS HOW WE DO SOMETHING EXACTLY n TIMES.

SOMETIMES WE START COUNTING AT ZERO

```
i=0;                    <- INITIALIZE LCV
while (i<n) {           <- TEST LRC
    // do something
    i++;                <- INCREMENT LCV
}
```

WE HAVE A SPECIAL CONSTRUCT WHICH PLACES THESE 3 STEPS IN THE SAME LOCATION IN THE PROGRAM.

```
for(i=1; i<=n; i++){
    // do something
}
```

OR IF YOU WANT i TO GO FROM 0 TO n-1 :

```
for(i=0; i<n; i++){
    //do something
}
```

Ex.

```
int i, n = 7
for(i=1; i<=n; i++)
    System.out.println( i*i);
System.out.println ("i=" +i);
```

NOTE THAT THE for loop is EXACTLY EQUIVALENT TO THE while loop CONSTRUCTION. IN PARTICULAR A TRACE OF THE VALUE OF i WOULD BE IDENTICAL AND THE loop is NOT EXITED UNTIL THE LRC is FALSE, i.e. i > n.

Ex.

```
// Average.java
Scanner sc = new Scanner(System.out);
double item, sum=0, average;
int count=0;
// Prompt for input
item = sc.nextDouble();
while (item != 0) {
    sum += item;
    count++;
    item = sc.nextDouble();
}
average = (count>0) ? (sum/count) : 0;
System.out.println (average);
```

THIS CAN ALSO BE DONE EQUIVALENTLY WITH A for loop.

```
// SAME DECLARATIONS
// Prompt for input
for(item=SC.nextDouble(); item!=0; item=SC.nextDouble()){
    sum += item;
    count++;
}
average = (count>0)? (sum/count) : 0;
// Print output.
```

Ex. Max.java

```
Scanner sc = new Scanner(System.in);
double item, max;

// Prompt for input
max = item = SC.nextDouble();
while (item != 0) {
    if (item > max)
        max = item;
    item = SC.nextDouble();
}
System.out.println(max);
```

EXERCISE: RE-WRITE THIS EXAMPLE AS A for loop.

THE SCOPE OF A VARIABLE IS THAT PART OF A PROGRAM WHERE IT CAN BE ACCESSED. IT EXTENDS FROM THE POINT WHERE IT IS DECLARED, TO THE END OF THE BLOCK IN WHICH THE DECLARATION TOOK PLACE.

Ex. // Scope.java

```
class Scope {
    Public static void main(String[] args) {
        int a = 6;
        System.out.println(a);
        {
            int b = 7;
            System.out.println(a);
            System.out.println(b);
        }
        System.out.println(a);
        System.out.println(b); //SYNTAX ERROR
    }
}
```

scope of a

scope of b

VARIABLES WITH THE SAME NAME MAY NOT HAVE OVERLAPPING SCOPE IN JAVA, AND A VARIABLE CANNOT BE REFERENCED OUTSIDE ITS SCOPE.

OFTEN A for loop is WRITTEN WITH A DECLARATION INSIDE ITS HEADING.

scope of i ↓

```
for(int i=0; i<10; i++) {
    // loop body to be
    // executed 10 times
}
// i CANNOT BE USED HERE UNLESS
// IT IS DECLARED AGAIN.
```

Ex // break & continue commands

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();

while(true){
   if(a>0){
        a=sc.nextInt();
        continue;
   }else if(a==0){
        break;
   }else{  // a<0
        break;
   } // continue goes here
} // break goes here,
System.out.println("done");
```

NOTE THAT break AND Continue WORK WITH do-while AND for loop STRUCTURES. Continue CAUSES THE Current ITERATION TO HALT AND THE NEXT ITERATION TO BEGIN. break CAUSES THE loop TO HALT, i.e. THE Current ITERATION IS FORCED TO BE THE LAST.

SEE OTHER EXAMPLES ON WEBSITE.

SEE P. 69-71 OF TEXT FOR ANOTHER TYPE OF CONTROL flow ALTERING STATEMENT, i.e. Switch.

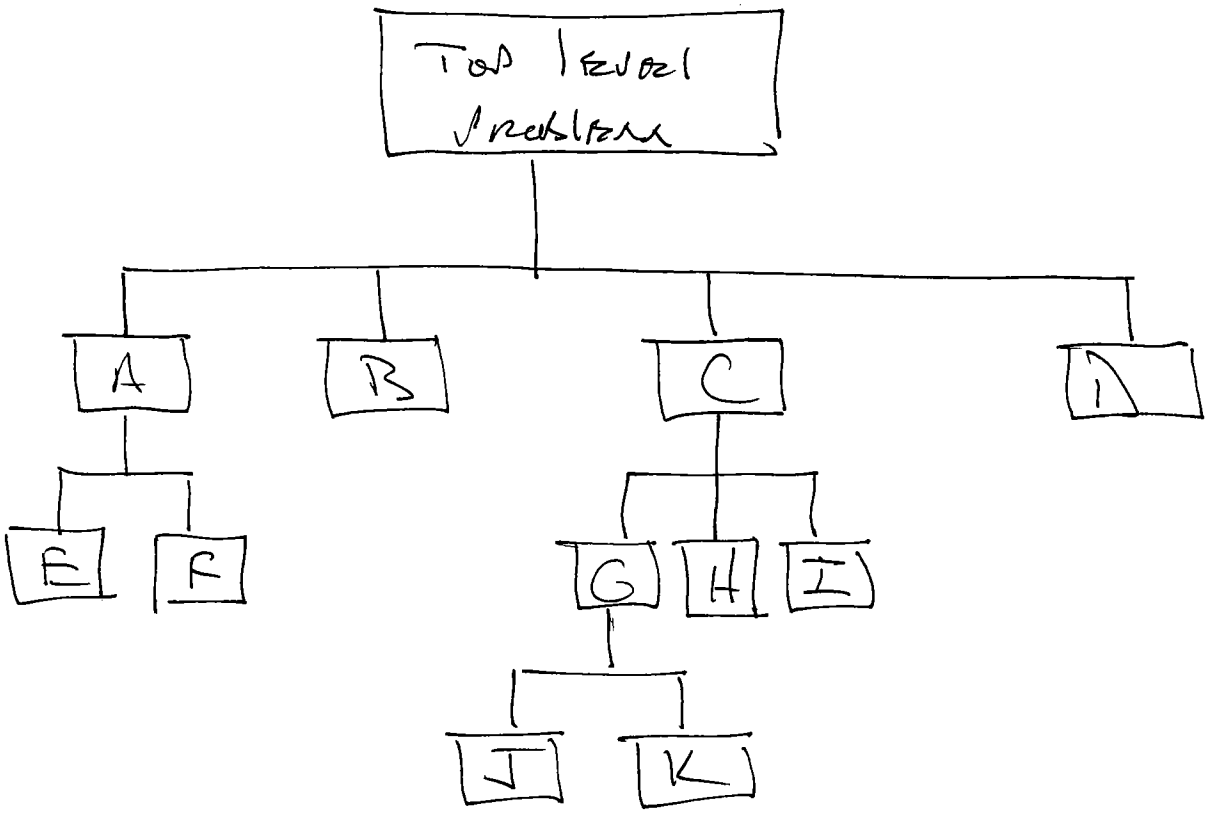
CHAPTER 4

METHODS & FUNCTIONAL ABSTRACTIONS

UP TO NOW WE'VE WRITTEN ONLY VERY SIMPLE PROGRAMS THAT DO VERY FEW THINGS. WE COULD BEGIN WRITING MORE COMPLEX PROGRAMS TO SOLVE VERY SOPHISTICATED PROBLEMS, BUT COMPLEXITY ITSELF POSES A SERIOUS PROBLEM. IT'S VERY DIFFICULT TO KEEP ALL THE DETAILS IN MIND, SO ERRORS AND INCONSISTENCIES OCCUR QUITE FREQUENTLY.

ONE WAY TO DEAL WITH THE PROBLEM OF PROGRAM COMPLEXITY IS VIA TOP-DOWN DESIGN, ALSO CALLED STEPWISE REFINEMENT. BASICALLY, THIS MEANS BREAKING THE OVERALL PROBLEM DOWN INTO A HIERARCHY OF SUBPROBLEMS, WHICH AT THE BOTTOM LEVEL, ARE VERY SIMPLE, THEN ASSEMBLE THE SUBPROBLEM SOLUTIONS INTO A SOLUTION TO THE OVERALL PROBLEM. THIS IS A CONCEPTUAL TECHNIQUE USED THROUGHOUT THE ENGINEERING DISCIPLINES TO DEAL WITH COMPLEXITY.

AT ANY GIVEN LEVEL OF THE HIERARCHY, ONE NEEDS BE CONCERNED WITH ONLY ONE SMALL PROBLEM. AS LONG AS THE RELATIONSHIPS BETWEEN THE VARIOUS SUB PROBLEMS ARE WELL DEFINED, IT IS POSSIBLE TO SOLVE VERY LARGE AND COMPLEX PROBLEMS THIS WAY.



WE OFTEN DRAW A STRUCTURE CHART AS ABOVE TO DIPICT THESE RELATIONSHIPS.

JAVA PROVIDES A CONSTRUCT CALLED A method (ALSO FUNCTIONS OR PROCEDURES) TO ACCOMPLISH THIS GOAL OF CREATING A HIERARCHY OF SUBPROBLEMS AND SOLUTIONS.

EX // HelloName.java
import java.util.*;

```
class HelloName {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String name;

        System.out.print("Enter name: ");
        name = sc.next();
        PrintMessage(name);
        System.out.println("bye");
    }

    static void PrintMessage(String s) {
        System.out.print("Hello ");
        System.out.print(name);
        System.out.print("!\n");
    }

}
```

The static method PrintMessage() APPEARS IN TWO CONTEXTS IN THIS PROGRAM.

- method call
- method definition

WHEN EXECUTION REACHES THE method CALL

```
PrintMessage(name);
```

IN main, MEMORY FOR THE method parameter s IS ALLOCATED, THE method ARGUMENT name IS COPIED TO THIS NEW memory s, THEN EXECUTION IS TRANSFERRED TO THE DEFINING BLOCK OF method PrintMessage(). WHEN THE last statement IS EXECUTED, EXECUTION RETURNS TO main RIGHT AFTER THE method call.

THE GENERAL FORM FOR A static method DEFINITION IS

```
static Type Name(Param. list) {
    // DEFINING Block
    return value; // NOT HERE IF
                  // TYPE is void
}
```

Ex

```
// Min.java
clas Min {
   Public static void main(String[] args){
      int a=6, b=7, m;
      m = min(a, b);
      System.out.print("min("+a+", ");
      System.out.println(b+")="+m);
   }

   static int min(int x, int y) {
      if (x < y)
         return x;
      else
         return y;
   }
}
```

- NOTE: REAL PROBLEMS RELATED TO SCOPE P.86-88

EXERCISE:
WRITE A METHOD CALLED max( ) WHICH TAKES TWO INTS AND RETURNS THE SMALLER OF THE TWO.

EX DOESN'T WORK!

```
public static void main(String[] args){
    int a=6, b=7;
    swap(a, b);
    System.out.println(a + " " + b);
}

static void swap(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

NOTE: a AND b ARE PASSED BY VALUE TO swap, WHICH MEANS THEIR VALUES ARE COPIED TO THE LOCAL VARIABLES