

Stepwise Refinement

Lecture 6
CGS 3416 Spring 2017

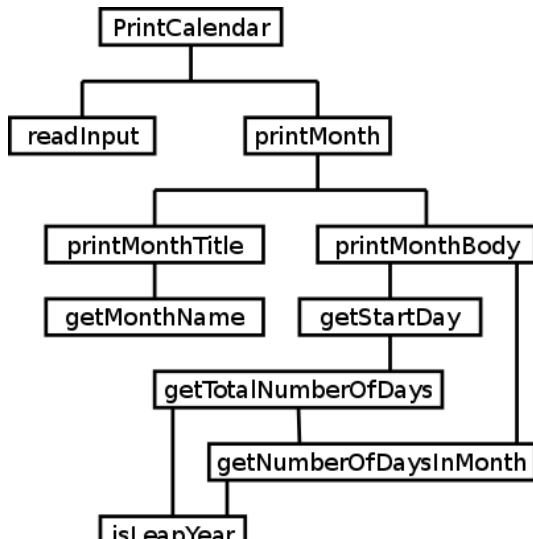
February 6, 2017

Programming is about Problem Solving

- Algorithm - a finite sequence of steps to perform a specific task
 - To solve a problem, you have to come up with the necessary step-by-step process before you can code it
 - This is often the trickiest part of programming
- Some useful tools and techniques for formulating an algorithm
 - Top-down Refinement: Decomposing a task into smaller and simpler steps, then breaking down each step into smaller steps, etc
 - Pseudocode: Writing algorithms informally in a mixture of natural language and general types of code statements
 - Flowcharting: If you can visualize it, it's often easier to follow and understand!

Top-down Refinement

Printing a calendar for any given month.



Example of a Pseudocode

Write down a pseudocode for the following problem:

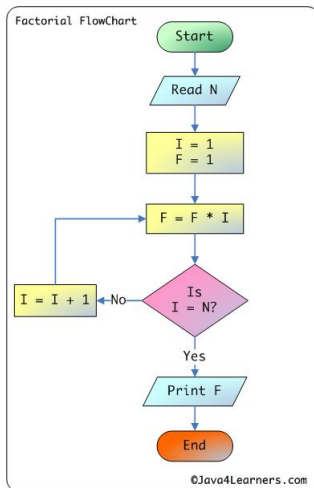
The program gets a number from the user and then if it is less than 100, it prints cheap; if it is less than 200, it prints acceptable, else it prints expensive.

- **Example Pseudocode:**

1. **begin**
2. **Input** price
3. **If** price < 100 then print “cheap”
4. **Else** if price < 200 then print “acceptable”
5. **Else** print “expensive”
6. **end**

Example of a Flow Chart

Write a program to compute factorial of a number in Java



Programming is about Problem Solving

- Testing - algorithms must also be tested!
 - Does it do what is required?
 - Does it handle all possible situations?
- Syntax vs. Semantics
 - Syntax – the grammar of a language.
A syntax error: "I is a programmer."
 - Semantics – the meaning of language constructs
Correct syntax, but a semantic error: "The car ate the lemur."

Top-Down Stepwise Refinement

Top down stepwise refinement is a useful problem-solving technique that is good for coming up with an algorithm.

Learning syntax is all well and good, but programming is really about problem-solving, and it takes practice.

Here's the general idea of thinking through an algorithm with stepwise refinement:

- Start with the initial problem statement
- Break it into a few general steps
- Take each "step", and break it further into more detailed steps
- Keep repeating the process on each "step", until you get a breakdown that is pretty specific, and can be written more or less in pseudocode
- Translate the pseudocode into real code

Example

Problem Statement: Determine the class average for a set of test grades, input by the user. The number of test grades is not known in advance (so the user will have to enter a special code – a “sentinel” value – to indicate that they’re finished typing in grades).

Initial breakdown into steps

- 1 Declare and initialize variables
- 2 Input grades (prompt user and allow input)
- 3 Compute class average and output result

Example

Now, breaking down the "compute" step further, we get:

Compute:

- ➊ add the grades
- ➋ count the grades
- ➌ divide the sum by the count

We realize this would be a problem, because to do all input before doing the sum and the count would require us to have enough variables for all the grades (but the number of grades to be entered is not known in advance). So we revise our breakdown of "steps".

Don't be afraid to go back and revise something if the initial plan runs into a snag!

Revised breakdown of steps

- 1 Declare and initialize variables
- 2 Input grades -- count and add them as they are input
- 3 Compute class average

Breaking the steps into smaller steps

So, now we can break down these 3 steps into more detail. The *input* step can roughly break down this way:

loop until the user enters the sentinel value

- 1 prompt user to enter a grade (give them needed info, like -1 to quit)
- 2 allow user to type in a grade (store in a variable)
- 3 add the grade into a variable used for storing the sum
- 4 add 1 to a counter (to track how many grades)

Further Refinement

We could specifically write this as a while loop or as a do-while loop. So one more refining step would be a good idea, to formulate the pseudo-code more like the actual code we would need to write. For example:

do

- 1 prompt user to enter a grade (give them needed info, like -1 to quit)
- 2 allow user to type in a grade (store in a variable)
- 3 add the grade into a variable used for storing the sum
- 4 add 1 to a counter (to track how many grades)

while user has NOT entered the sentinel value (-1 would be good)

Further Refinement

If we look at this format, we realize that the "adding" and "counting" steps should only be done if the user entry is a grade, and NOT when it's the sentinel value. So we can add one more refinement:

do

- 1 prompt user to enter a grade (give them needed info, like -1 to quit)
- 2 allow user to type in a grade (store in a variable)
- 3 if the entered value is a GRADE (not the sentinel value)

 add the grade into a variable used
 for storing the sum

 add 1 to a counter (to track how many
 grades)

while user has NOT entered the sentinel value (-1 would be good)

Some Notes on the breakdown

This breakdown helps us see what variables are needed, so the **declare and initialize variables** step can be now made more specific:

initialize variables:

- ① a grade variable (to store user entry)
- ② a sum variable (initialized to 0)
- ③ a counter (initialized to 0)

And the **compute answer** and print step becomes:

- ① divide sum by counter and store result
- ② print result

Putting it all together

initialize variables:

- ① a grade variable (to store user entry)
- ② a sum variable (initialized to 0)
- ③ a counter (initialized to 0)

Putting it all together

```
grade entry:
```

```
-----
```

```
do
```

- ① prompt user to enter a grade (give them needed info, like -1 to quit)
- ② allow user to type in a grade (store in a variable)
- ③ if the entered value is a GRADE (not the sentinel value)

```
    add the grade into a variable used  
    for storing the sum
```

```
    add 1 to a counter (to track how many  
    grades)
```

```
while user has NOT entered the sentinel value (-1 would be  
good)
```

Putting it all together

compute average:

- 1 divide sum by counter and store result
- 2 print result

What's left to do?

- It would be a good idea to refine the last step (compute and print average) more specifically, since it's possible for the user to type the sentinel value without entering any grades.
 - In this case, you don't want to divide by "counter", because that would be 0. This step should account for this possibility. (i.e. if the user has entered no grades, just print a message to that effect).
 - Otherwise, compute and print the average.
- Once the steps reach this level of detail, the pseudocode can be translated to real code. This is left as an exercise.