Problem Solving

Problem solving is writing an original program that performs a particular sett of tasks and meets all stated constraints.

The set of tasks can range from solving small coding exercises all the way up to building a social network site like Facebook or a search engine like Google. Each problem has its own set of constraints, for example, high performance and scalability may not matter too much in a coding exercise but it will be vital in apps like Google that need to service billions of search queries per day.

The best way to improve your problem solving ability is by building experience by making lots and lots of programs.

Understand the problem

The first step to solving a problem is understanding exactly what the problem is. If you don't understand the problem, you won't know when you've successfully solved it and may waste a lot of time on a wrong solution.

To gain clarity and understanding of the problem, write it down on paper, reword it in plain English until it makes sense to you, and draw diagrams if that helps. When you can explain the problem to someone else in plain English, you understand it.

Plan

Usually don't jump into code immediately after understanding the problem. You should plan out how you're going to solve it first. Some questions you should answer at this stage:

- Does your program have a user interface? What will it look like? What functionality will the interface have? Sketch this out on paper.
- What inputs will your program have? Will the user enter data or will you get input from somewhere else?
- What's the desired output?
- Given, your inputs, what are the steps necessary to return the desired output? What is the algorithm needed to solve the problem?

Pseudocode

Example pseudocode to print all numbers up to an inputted number:

```
When the user inputs a number
Initialize a counter variable and set its value to zero
While counter is smaller than user inputted number increment the counter by
one
Print the value of the counter variable
```

Divide and conquer

From your planning, you should have identified some subproblems of the big problem you're solving. Each of the steps in the algorithm we wrote out in the last section are subproblems.

You might not know all the steps that you need up front, so your algorithm may be incomplete. You see more of what you need to do as you progress working through the problem. Finding the next subproblem is easier if you have the first subproblem solved.

Solving Fizz Buzz

Fizz Buzz Example: "Write a program that takes a user’s input and prints the numbers from one to the number the user entered. However, for multiples of three print Fizz instead of the number and for the multiples of five print Buzz. For numbers which are multiples of both three and five print FizzBuzz."

Planning

Does your program have an interface? What will it look like? This is a browser console program so we don't need an interface. The only user interaction will be allowing the users to enter a number.

What inputs will your program have? Will the user enter data or will you get input from somewhere else? The user will enter a number from a prompt (popup box).

What's the desired output? The desired output is a list of numbers from 1 to the number the user entered. But each number that is divisible by 3 will output `Fizz`, each number that is divisible by 5 will output `Buzz` and each number that is divisible by both 3 and 5 will output `FizzBuzz`.

Writing the pseudocode

What are the steps necessary to return the desired output?

Example pseudocode:

```
When a user inputs a number
Loop from 1 to the entered number
If the current number is divisible by 3 then print "Fizz"
If the current number is divisible by 5 then print "Buzz"
If the current number is divisible by 3 and 5 then print "FizzBuzz"
Otherwise print the current number
```

Dividing and conquering

The first subproblem is getting input from the user:

```
let answer = parseInt(prompt("Please enter the number you would like to
FizzBuzz up to: "));
```

parseInt takes a string and returns an int (similar to + but different in some ways)

One way to tackle this in parts is to first make "Fizz" work, then make "Buzz" work, then finally make "FizzBuzz" work.

Assignment

"The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do."

Sometimes refactoring is the way to go.

How to Begin Thinking Like a Programmer

Coding is about 8 main concepts. Here are a few:

Learn how to use concepts in English
Write out concepts first, then convert to code later

Beginners think you don't understand what code to write
The real problem is not understanding the problem they are trying to solve

Comments are Code. Code explains the comments to the computer!

New variables:

- name - what do we call this thing?
- type - what type of data does it contain?
- initVal - what is it's starting value?

Output:

- message: text to write to user

Input:

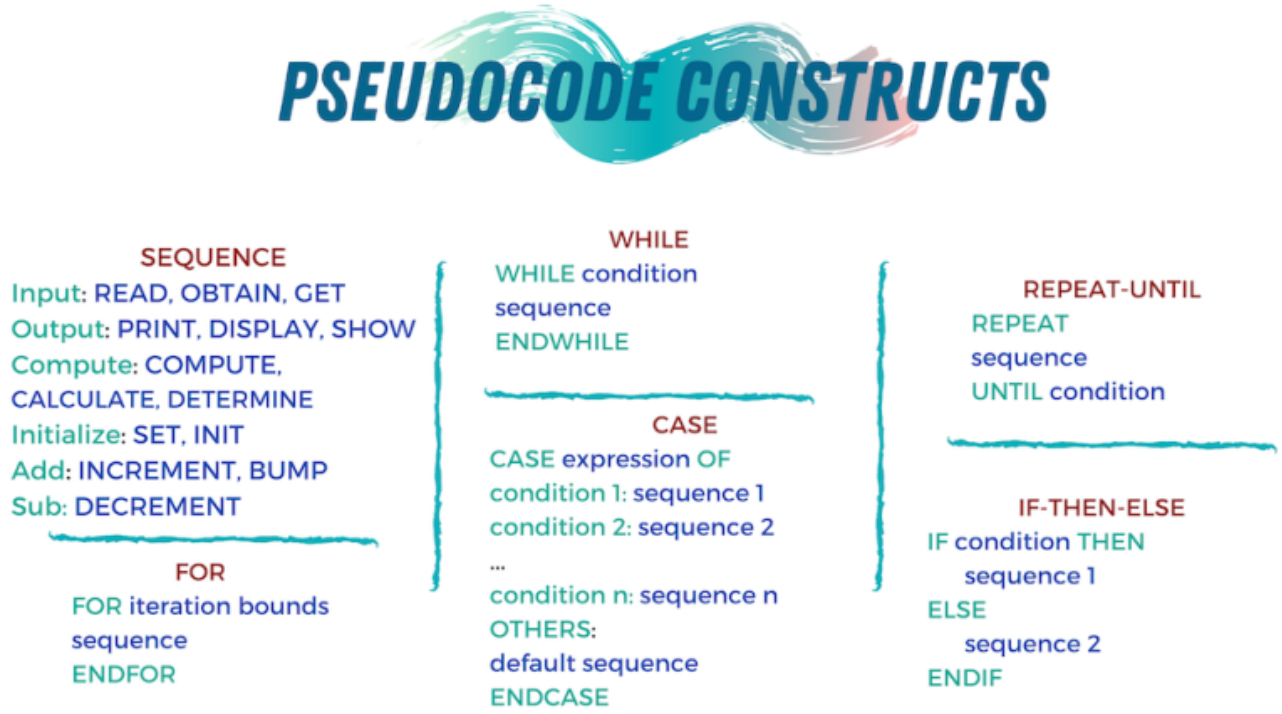
- variable - where answer from user will be stored
- message - question being asked of the user

Debugging: Did you tell the computer what to do incorrectly or did you tell it to do the wrong thing? Usually it's an algorithm problem not an implementation one. There's a problem with your understanding.

Pseudocode: What It Is and How to Write It

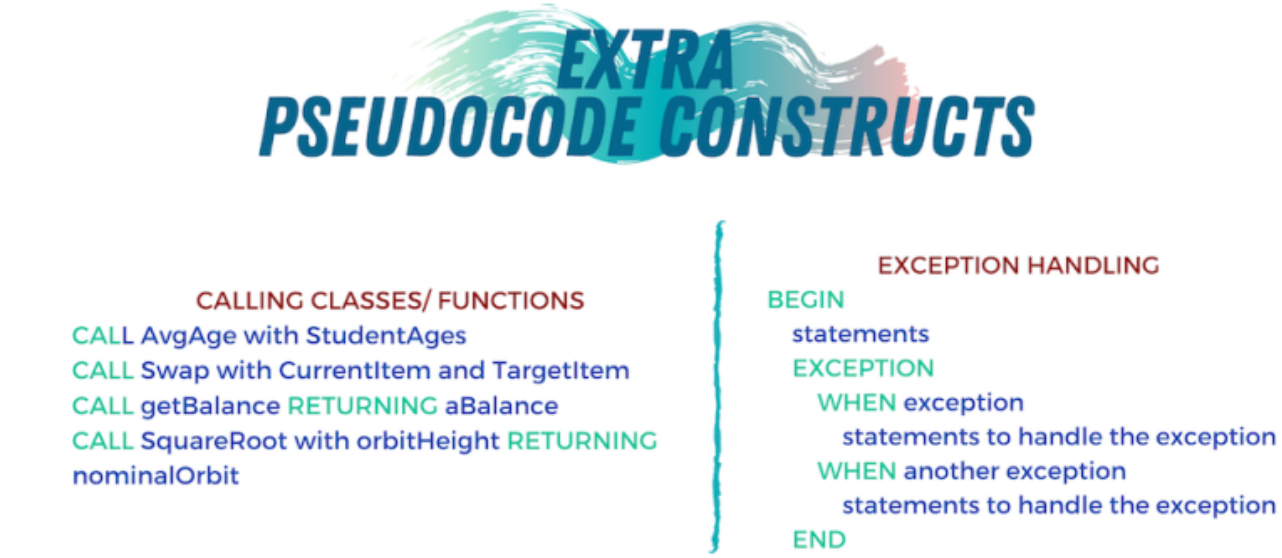
Pseudocode is the ability to represent six programming constructs (always written in uppercase): SEQUENCE, CASE, WHILE, REPEAT-UNTIL, FOR, and IF-THEN-ELSE. These describe the control flow of the algorithm while being language-agnostic:

1. **SEQUENCE** represents linear tasks sequentially performed one after the other.
2. **WHILE** is a loop with a condition at the beginning.
3. **REPEAT-UNTIL** is a loop with a condition at the bottom.
4. **FOR** is another way of looping.
5. **IF-THEN-ELSE** is a conditional statement changing the flow of the algorithm.
6. **CASE** is the generalization form of IF-THEN-ELSE.



Others include:

1. Invoking classes or calling functions (CALL keyword)
2. Handling exceptions (EXCEPTION, WHEN keywords)



However, pseudocode's rules are not written in stone and are less rigorous than those of a programming language.

Steps to write pseudocode:

1. Always capitalize the initial word (often one of the main six constructs).
2. Make only one statement per line.
3. Indent to show hierarchy, improve readability and show nested constructs.
4. Always end multi-line sections using any of the END keywords (ENDIF, ENDWHILE, etc.)
5. Keep your statements programming language independent.
6. Use the naming domain of the problem, not that of the implementation. For instance: "Append the last name to the first name" instead of "name = first + last."
7. Keep it simple, concise, and readable.

As the complexity and the size of a project increases, programmers come to realize how generating pseudocode makes writing the actual code much easier. Pseudocode helps you realize possible problems or design flaws in the algorithm earlier in the development stage, which saves you more time and effort on fixing bugs and avoiding errors down the road.

Why Use Psuedocode?

- It is easier to communicate algorithms to people of specialties beside coding.
- When the programmer goes through the process of developing and generating pseudocode, converting that into real code written in any programming language will become much easier and faster.
- Pseudocode is a good middle point between flowchart and code.
- Pseudocode is a helpful starting point for documentation. Sometimes, programmers even include the pseudocode as a docstring at the beginning of the code file.
- It can make debugging the algorithm itself easier. We can edit pseudocode more efficiently than testing, debugging, and fixing actual code.