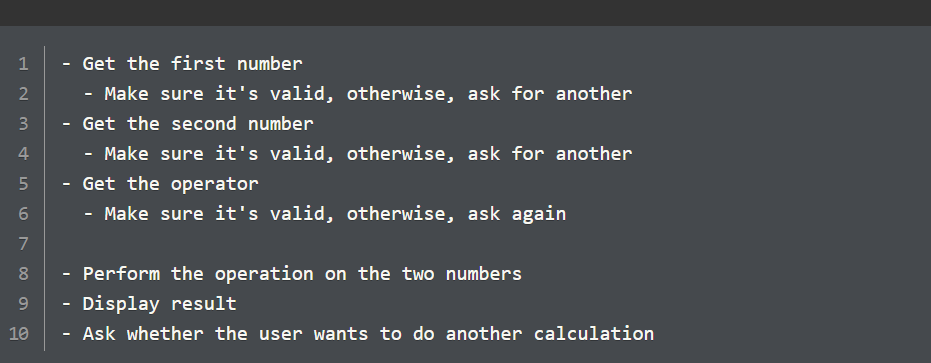
# External article: [Lessons from 1st half of course](https://medium.com/launch-school/lessons-from-the-first-half-of-core-50a6d85f9c76)

1. The 3 key phases of learning:
   1. **Exposure**: When you go through a lesson for the 1st time. The brain is familiarizing itself to the material rather than genuinely understanding it.
   2. **Learning and internalizing**: (Reviewing) You start to truly understand the material and develop accurate mental models.
   3. **Practice and externalizing**: Here you identify and correct minor errors in your mental models. You develop the skills to execute and clearly communicate the skills that you have learned so far.
2. 1st pass through the study material: Jot down the keywords and try to build a mental model by interconnecting them.
3. 2nd pass: A keen understanding of material followed by detailed notes.

# Writing Pseudocode

1. Instead of starting with trying to write code from the beginning, try to **load the problem in into your brain** first.
2. This takes effort, hence you must turn the problem around from all angles (logical analysis), before you even start thinking about the syntax.
3. Thus there are 2 layers to solving the problem:
   1. The logical problem domain layer
   2. Syntactical programming language layer
4. A lot of logic/assumption in a pseudocode can be incorrect (since it is a top-view solution). When that happens, you might have to restart writing it. This is alright since pseudocode is a guess at the solution and you cannot picture the solution fully until you translate it to code.
5. **Flowchart:** Helps to map out the logical sequence of a possible solution in a visual way.

|  |  |
| --- | --- |
|  | Flowchart for determining the largest number in a collection |

1. We include entire such codes as a node/subprocess within a larger process. We can than think of that node/subprocess as a zoomed out high-level view of this process given here.
2. For a high level view, we start with a declarative type syntax. For ex: Writing code for calculator-  
   
3. Notice, that you’re not providing info on how to validate the inputs. No specific or imperative code yet. Once you have this high level steps, you can then move on to the imperative pseudocode.  
     
   By not worrying about the low-level details of how these subprocesses work, we can think at a higher level about the overall application logic. Then, we can dive into each of those subprocesses and build detailed flowchart/pseudocode for each of them.

# Using ESlint

Running ESlint to verify your code for bugs/errors is an excellent habit. It also enforces style guidelines. Do the following steps for each project where you will need to run this file:

1. **npm install eslint@7.12.1 eslint-cli babel-eslint --save-dev** (Later versions are causing problems)
2. **npm WARN deprecated babel-eslint@10.1.0: babel-eslint is now @babel/eslint-parser. This package will no longer receive updates.**
3. **npx eslint -v** (check version)
4. **npx eslint** filename.js

# Debugging

1. Majority day-to-day life of a programmer is spent stuck on some problem.
   1. Coding is rather quick, majority time is spent analyzing and understanding a problem, experimenting, coming up with an approach or debugging bugs in the code.
   2. The key to debugging is a logical mind with patient temperament. Dealing with the sinking feeling and frustration, when something breaks is a critical aspect of learning to program.
2. **Stack trace:** Crucial in determining where to begin debugging. Carefully read the stack trace, it will also contain the error message, location of error as well as the trace to follow.
   1. Train your eyes to look for the relevant part of stack trace. Every language and library has a specific pattern to their stack trace.
   2. Study the error message and try to walk backwards through the code to understand how the program flow arrived at the error condition.
3. Steps to debugging:
   1. Reproducing the error.
   2. Determining the boundaries of the error.
   3. Trace the code and identify the source of error (AKA ‘Trapping the error’).
   4. Understanding the problem well.
   5. Implement a fix.
   6. Test the fix.
4. Dealing with multiple errors/bugs: You must attempt to fix only 1 error/bug at once. If you try to fix multiple errors at once, it is easy to get confused and make the problem even worse.  
   Instead, make a note of all the issues you come across and solve them, one at a time.
5. Techniques for debugging:
   1. Line-by-line: Developing a habit of reading your code line-by-line, char-by-char (with very high eye to detail) and have a lot of patience in dealing with errors.
   2. Rubber duck technique: Explaining your code in detail to an imaginary someone. Set yourself up to articulate the problem detail-by-detail.
   3. Walking away: Taking a break and returning with a fresh state of mind. This also gives time to your diffused mode of thinking to do its work.
   4. Using a debugger: Ex- Node inbuilt debugger
      1. node inspect filename
      2. exec varName
      3. n / next
      4. c / cont
      5. run / restart

# Debugging, preventing errors, catching errors

1. Only use **try/catch/finally** blocks if:
   1. A built-in fn/method can throw an exception/error and you need to handle or prevent that exception.
   2. A simple guard is impossible/impractical to prevent that exception.
2. Categories of errors:
   1. Source-code errors
   2. Edge-case errors
   3. Uncontrolled-input errors
3. Types of errors in JS:
   1. error: generic error (prototype of all the other errors)
   2. ReferenceError
   3. TypeError
   4. SyntaxError
   5. RangeError
   6. InternalError
   7. EvalError
   8. URIError

# Functions

1. Functions should either return a value or create a side-effect, but not both (except in few special cases).
2. Your goal should be to build small functions that are like LEGO blocks: they should be stand-alone pieces of functionality that you can use to piece together larger structures. You don't want these functions to be mentally taxing to use. Interesting structures comprise hundreds or thousands of individual LEGO pieces. Likewise, large programs combine hundreds or thousands of small functions.

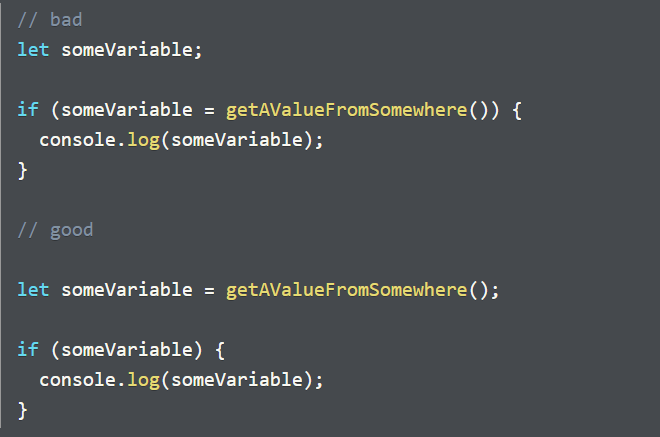
# Variable Scope

1. Variable shadowing-
   1. Variables in the inner scope prevents access of (or shadow) variables in the outer scope (that have the same name).
   2. Variables, parameters, function names, class names can all shadow names from outer scopes. The only names that do not shadow them from inside are property names for objects.
   3. You want to avoid shadowing at all cost. Giving descriptive variable names is a sure way to avoid shadowing.

# Are JS functions pass by value or pass by reference

1. pass by value: code inside the fn cannot mutate a global value
2. pass by reference: code inside the fn can mutate a global value
3. JS fn arguments are treated as pass by value when **primitive** values are passed to them, and these very arguments are treated as pass by reference when **objects** are passed to them.
4. Since JS fns can depict both these ways depending on the type of argument, it is combinedly called pass-by-value-of-the-reference (or call-by-sharing).
5. Remember, mutations can never occur with primitive values, but they can happen with most objects.
6. **Destructive fns/methods:** Fns/methods that mutate their caller are called destructive fns, ex: Array.push().

# Coding Tips 2

1. Never mutate a collection while iterating through it, or you might see unexpected results. However, you can mutate individual elements within the collection.
2. Avoid variable shadowing (choosing a local variable in the local scope with the same name as a variable in the outer scope).
3. Never use assignments within a conditional  
   
4. Coding is like writing, there are syntactical rules but there are also creative ways of expression.
5. Removing data from logic often makes the code more maintainable and scalable. Ex:

|  |  |
| --- | --- |
| **Comparison between the fn playerWins: when the data is separated vs. when the data is not separated** | |
|  |  |

The console will log the string “Hello” when the greeter function is executed, however in the case of greetiest function, it will throw a reference error. The reason for this error is that the function greetiest is defined inside the greeter function and is not accessible by the global scope where it is being executed. This code demonstrates the concept of variable scoping which says that inner scope can access variables from the outer scopes but outer scope (ex: global) cannot access variables (which includes functions) from inner scope (ex: greetiest function).