

# Algorithms Course Summary



Every day for at least one hour, we will challenge you with algorithm problems from our sequential curriculum.

## Daily Operation

Each onsite cohort will divide into groups, and groups will be tasked with solving algorithms on whiteboards. In the session's last 15 minutes, one group will present solutions, as will your instructor/TA. Online, challenges will be posted by the instructor at a common location.

## Goals

One important goal of the course is to get you comfortable describing your code's functionality. This is important for technical interviews, where you are asked to demonstrate your knowledge using only whiteboards.

Another important goal is to familiarize yourself with algorithms and data structures that solve complex problems efficiently, even as they scale worldwide.

## Rules

1. *Show up* – the only way to rewire your brain to think like a computer is *repetition*. Make sure you're on time for every session to get the challenge's introduction.
2. *No Laptops* – to simulate a technical interview, do not use laptop or refer to old code during challenges. Until directed otherwise, work problems on a whiteboard.
3. *Be respectful* – being able to walk others through your algorithm, explaining how it works, is as important as correctness. There will be many chances to discuss with peers or present to the class. All students should give the speaker full attention and respect. Public speaking is a common fear; we will learn to conquer our nerves. This only happens in a welcoming environment.
4. *Work in groups* – group work requires you to articulate your thoughts and describe your code, skills that are also useful when working in engineering teams more generally. Unless otherwise directed by the instructor, solve algorithm challenges in groups **no smaller than 2 and no larger than 3**.

## Presenting

Every day, two groups present solutions. Make sure all group members have a chance to describe the algorithm; give presenting groups the proper respect.

## Questions to ask about solutions

1. *Is it clear and understandable?*

Can you easily explain functionality and lead the listener through a T-diagram? Does the code self-describe effectively, or do you find yourself having to explain the meaning of the variables 'x' and 'i'?

2. *Is the output correct?*

Does your algorithm produce the required results? Is your algorithm resilient in the face of unexpected inputs or even intentional attempts to get it to crash?

3. *Is it concise?*

Remember the acronym DRY (Don't Repeat Yourself). Less code is better, so long as it is fully understandable. Pull any duplicate code into helper functions.

4. *Is it efficient?*

Does your function contain only necessary statements, and does it require only necessary memory? Does it stay efficient (in run time as well as memory usage), as the input size gets very large? Are you mindful of any intentional tradeoffs of time vs. space (improving run time by using more memory, or vice versa)?

## Tips

- Think out loud, to provide a window to your thinking. You may even get help if you are on the wrong track!
- Describe assumptions. Clarify before writing code.
- List sample inputs, along with expected outputs. This validates your understanding of the problem.
- Don't bog down. Add a comment, then move on.
- Break big problems down, into smaller problems.
- Focus on correct outputs, *then* on 'correct' solutions.
- Don't stress! Morning algorithms are brain cardio. This is not an evaluation of your abilities or expertise!
- Have fun! Consider these as simply puzzles that make you a better, more well-rounded developer.

# Table of Contents

---

<b>CHAPTER 0 – FOUNDATION CONCEPTS</b>	<b>1</b>
Computers, Software, and Source Code	1
Code Flow	2
Variables	3
Data Types	4
Not All Equals Signs Are the Same!	5
Printing to the Console	6
Conditionals	7
Functions	8
Combining Conditionals	9
Loops	10
FOR Loops	10
WHILE Loops	11
Other Loop Tips	12
Loops and Code Flow	13
Parameters	14
Foundations Review	15
<b>CHAPTER 1 – FUNDAMENTALS</b>	<b>16</b>
Return Values	17
Arrays	18
Writing Values into Arrays	19
Combining Arrays and FOR Loops	21
Using a T-Diagram	23
Comments	25
Fundamentals Review	26
<b>THE “BASIC 13”</b>	<b>27</b>
<b>CHAPTER 2 – FUNDAMENTALS, PART II</b>	<b>28</b>
Modulo Operator	29
Math Library	31
Using Modulo to Extract a Digit	32
Variables that Live Longer than a Single Function Call	33
Fundamentals Part II Review	35
<b>TESTING YOUR JAVASCRIPT CODE</b>	<b>36</b>
<b>CHAPTER 3 – ARRAYS</b>	<b>37</b>
Array.length	39
Passing By Reference	40
Another T-Diagram (Loops)	41
“Truthy” and “Falsey”	43
Arrays Review	44

<b>“BASIC 13” REVIEW</b>	<b>45</b>
<b>CHAPTER 4 – STRINGS AND ASSOCIATIVE ARRAYS</b>	<b>49</b>
More About Strings	49
Switch/case statements	51
Fast-Finish / Fast-Fail	53
Associative Arrays (Objects)	54
FOR ... IN Loops	55
Why Don’t We Allow Built-In Functions?	56
Strings and Associative Arrays Review	57
<b>CHAPTER 5 – LINKED LISTS</b>	<b>58</b>
Objects and Classes	58
Linked List	59
Prompts	62
Alerts	63
Write Understandable Code	66
<b>THE “BUGGY 13” (#1)</b>	<b>67</b>
<b>CHAPTER 6 – QUEUES AND STACKS</b>	<b>69</b>
Queues	70
Stacks	71
Circular Queues	73
<b>THE “BUGFUL 13” (#2)</b>	<b>76</b>
<b>CHAPTER 7 – ARRAYS, PART II</b>	<b>79</b>
Test-Driven Development	79
Divide and Conquer	80
Time-Space Tradeoff	82
Data Sufficiency	84
<b>THE “BUG-LADEN 13” (#3)</b>	<b>85</b>
<b>CHAPTER 8 – LINKED LISTS, PART II</b>	<b>88</b>
Runners and Linked List Iterators	89
Doubly Linked List	93
<b>THE “BUG-INFESTED 13” (#4)</b>	<b>95</b>
<b>CHAPTER 9 – RECURSION</b>	<b>98</b>
Three requirements for effective recursion	98
T-Diagrams and Recursion	100
Dynamic Programming and Memoization	102

<b>CHAPTER 10 – STRINGS, PART II</b>	<b>109</b>
<b>CHAPTER 11 – TREES</b>	<b>115</b>
Binary Tree Depth	117
Binary Search Tree Traversal	118
Self-Instantiating Classes	119
Making BST a Fully Navigable Data Structure	121
<b>CHAPTER 12 – SORTS</b>	<b>123</b>
Big-O Notation	125
Adaptivity	126
Stability	127
Memory Analysis	128
Sorting Review	130
<b>CHAPTER 13 – SETS AND PRIORITY QUEUES</b>	<b>131</b>
Sets and Multisets	131
Set Operations	132
Set Theory Recap	134
Priority Queues	135
Heap Data Structure	136
<b>CHAPTER 14 – HASHES</b>	<b>140</b>
Hash Collisions	142
<b>CHAPTER 15 – TREES, PART II</b>	<b>147</b>
Full Trees and Complete Trees	147
Repairing a Binary Search Tree	148
BST Partitioning	149
Repairing a More Complex Binary Search Tree	150
Breadth-First Search	151
<b>CHAPTER 16 – TRIES</b>	<b>152</b>
Trie Data Structure	152
Trie MultiSet	155
Trie Map	155
<b>CHAPTER 17 – GRAPHS</b>	<b>156</b>
Graph Terms	157
Representing Graphs	158
Edge List	159
Adjacency Map	160
Adjacency List	161
Directed and Undirected Graphs	162
Edges Have Weight	162
Depth-First Search	163
Breadth-First Search	164

Directed Acyclic Graphs	165
<b>CHAPTER 18 – BIT ARITHMETIC</b>	<b>166</b>
Numerical Systems	166
Octal System	167
Hexadecimal System	168
Binary System	169
Bitwise Operators, Part 1	170
Bitwise Operators, Part 2	171
Bit Shifting and Masking	172
<b>CHAPTER 19 – TREES, PART III</b>	<b>174</b>
AVL Trees	174
Rotation	176
Red-Black Trees	179
Splay Trees	180
<b>CHAPTER 20 – SPATIAL, LOGIC, ESTIMATION</b>	<b>182</b>
Estimation	185
<b>CHAPTER 21 – OPTIMIZATION</b>	<b>186</b>
The Performance Journey	186
Code Tuning	206
Optimization Review	217
<b>INDEX OF CHALLENGES</b>	<b>218</b>
<b>INTERVIEW TIPS</b>	<b>230</b>

## Computers, Software, and Source Code

Computers are amazing. They can rapidly perform complex calculations, store immense amounts of data, and almost instantly retrieve very specific bits of information from this mountain of data they have stored. In addition to being really fast, they sometimes appear almost superhuman in their ability to use information to make decisions. How do they do it?

Simply put, computers are machines. We as humans are skilled at creating tools that perform *specific* tasks very well: a toaster, for example, or a lawn mower. Computers are tools built from components such as semiconductor *chips*, and ongoing advances in material science enable these pieces to get smaller and faster with every successive year (see *Moore's Law*). This is why computers have become so breathtakingly fast, but it only partially explains why they can seem so *smart*. It doesn't really tell us why they are so universally relied upon to solve such a diverse range of problems across today's world.

Computing is exciting because computing devices are flexible – they can be *told* to do things not imagined when they were originally built. Science fiction may become science in the future, but for today they only know what we *tell* them; they only do as they are told. Who *teaches* them, who *tells* them what to do? Software engineers, developers, programmers! You are reading this, so you probably intend to be one too. From a computer's viewpoint, you are training to become an educator. (-:

How do programmers tell computers what to do? We create **Software**. Software is a sequence of instructions that we build and provide to a computer, which then "mindlessly" runs those instructions. Computers cannot natively understand human language, nor can humans read the language of semiconductor components. To talk with computers, we need a "go-between" format: something that software engineers can understand, yet can also be translated into machine language instructions.

Many of these "go-between" languages have been created: PHP, Python, Ruby, JavaScript, Swift, C#, Java, Perl, Erlang, Go, Rust, and others – even HTML and CSS! Each language has differing strengths and therefore is useful in different situations. Programming languages all do essentially the same things though: they read in a series of human-readable steps instructing a computer how to respond, and they translate the steps into a format the computer can understand and later execute.

All these sequences of instructions, written in programming languages like JavaScript, are what we call **Source Code**. How and when this source code is translated into *machine code* will depend on the language and the machine. *Interpreted languages* like PHP, Python and Ruby translate from source code into machine code "on the fly", immediately before a computer needs it. *Compiled languages* do some or all of this translation ahead of time. As we said earlier, a computer simply follows instructions it was given. More specifically, though, it executes (*runs*) machine code that was built (*translated*) from some piece of source code: code that was written by a software engineer.

Let's teach you how to think like a computer, so you can write effective source code. We have chosen to use JavaScript as the programming language for this book, so along the way you'll learn the specifics of that language. With very few exceptions, however, the concepts are universal.

# **Chapter 0 – Foundation Concepts**

---

## **Code Flow**

When a computer executes (runs) a piece of code, it simply reads each line from the beginning of the file, executing it in order. When the computer gets to the end of the lines to execute, it is finished with that program. There may be *other* programs running on that computer at the same time (e.g. pieces of the operating system that update the monitor screen), but as far as your program goes, when the computer's execution gets to the end of the source you've given it, the program is done and the computer has completed running your code.

It isn't necessary for your code to be purely linear from the top of the file to the end of the file, however. You can instruct the computer to execute the same section of code multiple times – this is called a program **LOOP**. Also, you can have the computer jump to a different section of your code, based on whether a certain condition is true or false – this is called an **IF-ELSE** statement (or a conditional). Finally, for code that you expect to use often, whether called by various places in your own code, or perhaps even called by others' code, you can separate this out and give it a specific label so that it can be called directly. This is called a **FUNCTION**. More on these later.

# **Chapter 0 – Foundation Concepts**

---

## **Variables**

Imagine if you had two objects (a book and a ball) that you wanted to carry around in your hands. With two hands, it is easy enough to carry two objects. However, what if you also had a sandwich? You don't have enough hands, so you need one or more containers for each of the objects. What if you had a box with a label on it, inside which you can put one of your objects. The box is closed, so all you see is the label, but it is easy enough to open the box and look inside. This is essentially what a *variable* does.

A variable is a specific spot in memory, with a label that you give it. You can put anything you want into that memory location and later refer to the value of that memory, by using the label. The statement:

```
var myName = 'Martin';
```

creates a variable, gives it a label of `myName`, and puts a value of "`Martin`" into that memory location. Later, to reference or inspect the value that you stored there, you simply refer to the label `myName`. For example (jumping ahead to the upcoming [Printing to the Console](#) topic), to *display* the value in the variable `myName`, do this:

```
console.log(myName);
```

# Chapter 0 – Foundation Concepts

---

## Data Types

Containers exist to hold things. You would create a variable because you want it to store some value – some piece of information. A value could be a number, or a sentence made of text characters, or something else. Specifically, JavaScript has a few *data types*, and all values are one of those types. Of these six data types, three are important to mention right now. These are Number, String, and Boolean.

In JavaScript, a *Number* can store a huge range of numerical values from extremely large values to microscopically small ones, to incredibly negative values as well. If you know other programming languages, you might be accustomed to making a distinction between integers and floating-point numbers – JavaScript makes no such distinction.

A *String* is any sequence of characters, contained between quotation marks. In JavaScript you can use either single-quotes or double-quotes. Either way, just make sure to close the string the same way you opened it. '`Word`' and "`wurd`" are both fine, but '`weird`' and "`whoa!`" are not.

Finally, a *Boolean* has only two possible values: `true` and `false`. You can think of a Boolean like a traditional light switch, or perhaps a yes/no question on a test. Just as a light switch can be either *on* or *off*, and just as a yes/no question can be answered with either *yes* or *no*, likewise a Boolean must have a value of either `true` or `false` – there is nothing in-between.

One of the main things we do with variables, once they contain information of a certain data type, is to compare them. This is our next section.

# Chapter 0 – Foundation Concepts

---

## Not All Equals Signs Are the Same!

In many programming languages, you will see both `=` and `==`. These mean two different things! The code `(x = y)` can be described as “Set the value of X to become the value of Y”, while `(x == y)` can be described as “Is the value of X equivalent to the value of Y?” It is more common – but less helpful right now when you are learning these concepts – to hear these verbalized as “Assign Y to X” and “Are X and Y equal?”, respectively.

In short, `=` sets things, and `==` tests things. Said another way, *single-equals is for assignment, and double>equals is for comparison*.

Many programming languages are extremely picky about data types. When asked to combine two values that have differing data types, some languages will halt with an error rather than do so. JavaScript, however, is not so strict. In fact, you could say that JavaScript is very *loosely* typed: it very willingly changes a variable’s data type, whenever needed. Remember the `==` operator that we described previously? It actually means “after converting X and Y to the same data type, are their values equivalent?” If you want *strict comparison* without converting data types, use the `===` operator.

Generally, `==` is advised, unless you explicitly intend to equate values of differing types, such as `[1, true, "1"]` or `[0, false, "0"]`, etc. (If this last sentence doesn’t make sense yet, don’t worry.)

Quick quiz:

- How many values are accepted by the `==` and `===` operators?
- Do inputs to the `==` and `===` operators need to be of the same data type?
- What is the data type returned by the `==` and `===` operators?

Hey! Don’t just read on: jot down answers for those questions before moving on. Done? OK, good....

Answers:

- The `==` and `===` operators both accept two values (one before the operator, and one after).
- No, the two values need not be the same type for any of these operators. However, if they are not the same type, `==` always returns `false`. The `==` operator internally converts values to the same type before comparing.
- The `==` and `===` operators both return a Boolean value.

Now that you know just a little about Numbers, Strings and Booleans, let’s start using them.

# Chapter 0 – Foundation Concepts

---

## Printing to the Console

Eventually, you will create fabulous web systems and/or applications that do very fancy things with graphical user interface. However, when we are first learning how to program, we start by having our programs write simple text messages (strings!) to the screen. In fact, the very first program that most people write in a new language is relatively well known as the “Hello World” program. In JavaScript, we can quickly send a text string to the *developer console*, which is where errors, warnings and other messages about our program go as well. This is not something that a normal user would ever look at, but it is the easiest way for us to print variables or other messages.

To log a message to the console, we use `console.log()`. Within the parentheses of this call, we put any message we want displayed. The `console.log` function always takes in a string. If we send it something that isn’t a string, JavaScript will first convert it to a string that it can print. It’s very obliging that way. So our message to be logged could be a literal value (42 or "Hello"), or a variable like this:

```
console.log("Hello World!");

var message = "Welcome to the Dojo";
console.log(message);
```

We can also combine literal strings and variables into a larger string for `console.log`, simply by adding them together. If you had a variable `numDays`, for example, you could log a message like this:

```
var numDays = 40;
console.log("It rained for " + numDays + " days and nights!");
```

Notice that we put a space at the end of our "It rained for " string, so that the console would log "It rained for 40 days and nights!" instead of "It rained for40days and nights!"

So, what would the following code print?

```
var greeting = "howdy";
console.log("greeting" + greeting);
```

It would print "greetinghowdy", since we ask it to combine the literal string "`greeting`" with the value of the variable `greeting`, which is "howdy". Make sense?

One last side note: if you *really insist* upon writing a string to the actual web page, you could use the old-fashioned function `document.write()`, such as `document.write("Day #" + numDays)`. However, you won’t use this function when creating real web pages, so why get into that habit now?

# Chapter 0 – Foundation Concepts

---

## Conditionals

If you are driving and get to a fork in the road, you have to make a decision about which way to go. Most likely, you will decide which way to go based on some very good reason. In your code, there is a mechanism like this too. You can use an **IF** statement to look at the value of a variable or perhaps compare two variables, and execute certain lines of code based on that result. If you wish, you can also execute *other* lines of code if the result goes the other way. The important point here is that each decision has only two possible outcomes. You have a certain test or comparison that is done, and **if** that test passes, then you execute certain code. If you wish, you can also have a different set of code that you execute in the “*test did not pass*” case (called the **ELSE**). This code would look like this:

```
if (myName == "Martin")
{
    console.log("Hey there Martin, how's it going?");
}
```

The **IF** statement is followed by parentheses that contain our *test*. Remember that we need to use a double-equals to create a comparison, and here we are comparing the value of variable **myName** to the string **"Martin"**. If that comparison passes, then we will execute the next section of code, within the curly braces. Otherwise, we will skip those lines of code. What if we want to greet the user with some other cheerful comment, even if the user was not Martin? If you wish, you can also have a different set of code that you execute in the “*test did not pass*” case (called the **ELSE**). We might write code like this:

```
if (myName == "Martin") {
    console.log("Hey there Martin, how's it going?");
} else {
    console.log("Greetings Earthling. Have a great day!");
}
```

Two other notes. First, the “*test*” between the parentheses for these **IF** statements is an expression resulting in a *Boolean*. It can be more than a single comparison. You can use a combination of logical AND, OR and NOT connectors that still result in a Boolean for the **IF** to evaluate. Second, we can nest **IF..ELSE** statements inside other **IF..ELSE** statements, or even chain them together like this:

```
if (myName == "Martin") {
    console.log("Hey there Martin, how's it going?");
} else if (myName == "Beth") {
    console.log("You look fabulous today!");
} else {
    console.log("Greetings Earthling. Have a great day!");
}
```

Let’s explore both of these ideas further on the next few pages.

# Chapter 0 – Foundation Concepts

---

## Functions

Let's say that you are writing a piece of code that has five different places where it needs to print your name. As mentioned above, for code that you expect to call often, separate this out into a different part of your file, so these lines of code don't need to be duplicated each time you print your name. This is called a **FUNCTION**. Creating (or *declaring*) a function could look like this:

```
function sayMyName( )
{
    console.log("My name is Martin");
}
```

By using the special **function** word, you tell JavaScript that what follows is a set of source code that can be called at any time by simply referring to the **sayMyName** label. Note: *the code above does not actually call the function immediately*; it sets the function up for other code to use (call) it later.

'Calling' the *function* is also referred to as 'running' or 'executing' the function. If the above is how you declare a function, then the below is how you actually *run* that function:

```
sayMyName();
```

That's it! All you need to do is call that label, followed by open and close parentheses. The parentheses are what tell the computer to execute the function with that label, so don't forget those.

One last thing: there is nothing stopping a function from calling other functions (or in certain special situations, even calling itself!). You can see above that the **sayMyName** function does, in fact, call the built-in **console.log** function. Naturally, you would not expect **console.log** to run *until* you actually called it; in the same way, any code that you write will only start running when some other code calls it (maybe part of the computer browser or the operating system). So, except for the very first piece of code that runs when a hardware device starts up, all other code runs only because other code called it.

To review: when we *declare* a function, it allows some other *caller* to execute our function from some other place in the code, at some other time. It does not run the function immediately. So if your source code file contains this:

```
function sayMyName( )
{
    console.log("My name is Martin");
}
```

...and then you execute that source code file, nothing will actually appear in the developer console. This is because no code ever called your **sayMyName** function. You set it up, but you never used it.

# Chapter 0 – Foundation Concepts

---

## Combining Conditionals

In JavaScript, we can create compound conditionals such as “*If it is Friday and I’m in a good mood, let’s go have fun!*” Hence we would not go if either it is not Friday, or if I’m not in a good mood.

**AND** operators combine two logical tests, requiring **both** true for result to be true. Logical AND is double-&, located between two logical conditions. The above could be represented in source this way:

```
if (today == "Friday" && moodLevel >= 100) { goDancing(); }
```

As conveyed, we need both to be true. If not Friday or if mood less than 100, we don’t go.

**OR** operators combine two logical tests, returning true if **either** is true. An example might be “*If it is raining or too far to walk, let’s call Uber!*” The logical OR is double-|, located between logical conditions:

```
if (raining == true || distanceMiles > 3) { callUber(); }
```

We run `callUber()` unless **both** tests are not true (i.e. not raining and distance is three or less).

**NOT** operators invert a single boolean: true becomes false, and false becomes true. Logical NOT is the exclamation point !. “*If it isn’t snowing, I’ll wear shorts*” could become this source code:

```
if (!snowing) { bravelyDonSomeShorts(); }
```

The function would be called only when **IF** statement was **true**, which is when `!snowing` is **true**, which is when `snowing` is **false**. Can you decipher the below? When would you walk / fly / swim?

```
if (weather != "rainy") {
    if (distanceToStadium < 3) {
        console.log("I think I'll walk to the game.");
    } else {
        console.log("It's a bit far, so maybe I'll fly.");
    }
} else {
    console.log("Hey, I'm a duck! A little water is OK. I'll swim.");
}
```

If not rainy, and distance to stadium is less than 3 (miles?), we walk. If distance is 3 or more, and weather is not rainy, we fly. If weather is rainy, regardless of distance, we swim. What an odd duck!

```
if (CONDITION_1 && CONDITION_2) { // '&&' means 'AND'
    // body of 'IF': runs if CONDITION_1 is true AND CONDITION_2 is true
}
else {
    // 'ELSE' body: runs if CONDITION_1 is false OR CONDITION_2 is false
}
```

# Chapter 0 – Foundation Concepts

---

## Loops

Sometimes you will have lines of code that you want to run more than once in succession. It would be very wasteful to simply copy-and-paste that code over and over. Plus, if you ever needed to change the code, you would need to change all those lines one by one. What a mess! Instead, you can indicate that a section of code should be executed some number of times. Consider the following: “Do the next thing I tell you *four* times: hop on one foot.” That would be much better than “Hop on one foot. Hop on one foot. Hop on one foot. Hop on one foot.” (even though it is just as silly) Programming languages have the concept of a *LOOP* that is essentially a section of code that will be executed a certain number of times. There are a few different types of loops. The most common are **FOR** and **WHILE** loops. Shall we explore each of them in turn? Yes, let’s.

## FOR Loops

**FOR** loops are good when you know exactly how many times those lines of code should be run. **WHILE** loops are a combination of loops with conditionals; they are good when you don’t know how many times to loop, but you know you want to loop *until* a certain test is true (or more accurately, if you want to continue looping *while* a certain test continues to be true). To create a **FOR** loop, in addition to the chunk of code to be looped, you need to specify three things, and these things go into the parentheses after the **FOR**: any initial setup, a test that needs to be true in order to continue looping, and any code that should be run at the end of each time through the loop. Here is an annotated example:

```
//      A          B          D
for (var num = 1; num < 6; num = num + 1)
{
    console.log("I'm counting! The number is ", num);    // C
}
console.log("We are done. Goodbye world!");                // E
```

The above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

How does this **FOR** loop really work? Up front, the local variable **num** is created and is set to a value of 1. This step A happens exactly once, and then our loop starts. Step B: **num** is compared to 6. If it is less than 6, then (step C) the chunk of code within the curly braces is executed, and after that (step D) 1 is added to **num**. Then we go back to step B. When the test at B fails for the first time, we immediately exit without executing the code in C or D. At that point, code execution continues onward from E, the point immediately following the close-curly-brace. Or, said another way:

```
for (INITIALIZATION; TEST; INCREMENT/DECREMENT) {
    // BODY of the loop - this runs repeatedly while TEST is true
}
// INIT. [TEST?-BODY-INCREMENT] (repeatedly while TEST is true). Exit.
```

# Chapter 0 – Foundation Concepts

---

## WHILE Loops

**WHILE** loops are similar to **FOR** loops, except with two pieces missing. First, there is no upfront setup like is built into a **FOR** loop. Also, unlike a **FOR** statement, a **WHILE** doesn't automatically include code that is executed at the end of each loop (our D above). **WHILE** loops are great when you don't know how many times (iterations) you will loop. Any **FOR** loop can be written as a **WHILE** loop. For example, the above **FOR** loop could be written instead as this **WHILE** loop, which would execute identically:

```
var num = 1;                                // A
while (num < 6)                            // B
{
    console.log("I'm counting! The number is " + num); // C
    num = num + 1;                           // D
}
console.log("We are done. Goodbye world!"); // E
```

Behaving identically with the above **FOR** loop, the **WHILE** code written immediately above will execute in this sequence: A - B-C-D - B-C-D - B-C-D - B-C-D - B - E.

Let's review before we move on. Anything we do with a **FOR** loop, we could do with a **WHILE** loop instead – and vice versa. So when should we use **FOR** loops, and when should we use **WHILE** loops? Generally, use **FOR** loops when you know *exactly* how long a loop should run. Use **WHILE** loops when you have a condition that keeps the loop running (or that will cause the loop to stop), but you aren't sure exactly how many iterations the loop will require.

# Chapter 0 – Foundation Concepts

---

## Other Loop Tips

Some developers like to increment a variable's value by running `num += 1`; this is the same as typing `num = num + 1`. You may sometimes see `num++` or even `++num`; both are equivalent to `+=`.

```
var index = 2;
index = index + 1;
index++;
// index now holds a value of 4
```

By that same token, we can decrement the value of num by running simply `num--`; or `--num`. This is exactly the same as running `num = num - 1`; or `num -= 1`. There are `*=` and `/=` operators as well, that multiply and divide a number as you might expect.

```
var counter = 5;
counter = counter - 1; // counter now holds a value of 4
counter--;           // counter is now 3
counter *= 6;         // counter is 18
counter /= 2;         // counter == 9
```

Furthermore, not every loop must increment by one. Can you guess what the following would output?

```
for (var num = 10; num > 2; num = num - 1) {
  console.log('num is currently', num);
}
```

Yes that's right: this `FOR` loop counts backwards by ones, starting at 10, while `num` is greater than 2. So it would count `10, 9, 8, 7, 6, 5, 4, 3`.

How would you print all even numbers from 1 to 1000000? How would you print all the *multiples of 7* (7, 14, ...) up to 100? Understanding how to use `FOR` loops is critical, so get really familiar with this.

# Chapter 0 – Foundation Concepts

---

## Loops and Code Flow

With more complex loops, you might discover that you need to break out of a loop early, or instead to skip the rest of the current pass but continue looping. In JavaScript, there are the special **BREAK** and **CONTINUE** keywords that allow you to do this. If you add **break;** to your code, program execution will *immediately exit the specific loop you are currently in*, and will continue executing immediately following the loop. Even the final end-loop statement (`num = num + 1` above) will not be executed. If you add **continue;** to your code, the rest of that pass through the loop *will be skipped* but the loop-end statement *is executed* and looping will continue. With both of these, once that statement runs, any subsequent code in that loop will be skipped. See examples below.

The following code will print the first two lines, but then will immediately exit the while loop.

```
var num = 1;
while (num < 5) {
  if (num == 3) {
    break;
    // if you have additional code down here, it will never run!
  }
  console.log("I'm counting! The number is ", num);
  num = num + 1;      // if we break, these lines won't run
}
```

*I'm counting! The number is 1*

*I'm counting! The number is 2*

The following code will count from 1 to 4, printing a statement about each number, but will completely forget to say anything about 3, because when `num == 3`, the **continue** forces it to skip the rest of that loop and continue from the top of the loop (after incrementing `num`).

```
for (var num = 1; num < 5; num += 1) {
  if (num == 3) {
    continue;
    // if you have additional code down here, it will never run!
  }
  console.log("I'm counting! The number is ", num);
}
```

*I'm counting! The number is 1*

*I'm counting! The number is 2*

*I'm counting! The number is 4*

**Continue** is uncommon, but you'll see **break** everywhere. Get comfortable setting up loops that (if things go well) execute for a certain number of iterations, but at any time could exit if you encounter a certain condition. With **break**, it isn't preposterous to see **while(true)** in your code! My goodness.

# Chapter 0 – Foundation Concepts

---

## Parameters

Being able to call another function can be helpful for eliminating a lot of duplicate source code. That said, a function that always does exactly the same thing will be useful only in specific situations. It would be better if functions were more flexible and could be customized in some way. Fortunately, you can pass values into functions, so that the functions can behave differently depending on those values. The caller simply inserts these values (called *arguments*) between the parentheses, when it executes the function. When the function is executed, those values are copied in and are available like any other variable. Specifically, inside the function, these copied-in values are referred to as *parameters*.

For example, let's say that we have pulled our friendly greeting code above into a separate function, named `greetSomeone`. This function could include a parameter that is used by the code inside to customize the greeting, just as we did in our standalone code above. Depending on the argument that the caller sends in, our function would have different outcomes. Tying together the ideas of functions, parameters, conditionals and printing, this code could look like this:

```
function greetSomeone(person)
{
    if (person == "Martin") {
        console.log("Yo dawg, howz it goin?'");
    }
    else {
        console.log("Greetings Earthling!");
    }
}
```

You might notice in the code above that there are curly braces that are not alone on their own lines, as they were in the previous code examples. The JavaScript language does not care whether you give these their own line or include them at the end of the previous line, as long as they are present. Really, braces are a way to indicate to the system some number of lines of code that it should treat as a single group. Without these, **IF..ELSE** and **WHILE** and **FOR** statements will only operate on a single line of code. Even if your loop *is* only a single line of code (and hence would work without braces), it is always safer to include these, in case you add more code to your loop later – and to reinforce good habits.

# Chapter 0 – Foundation Concepts

---

## Foundations Review

Hopefully, this first chapter made you more comfortable with the essential building blocks of software. Below is a summary of the ideas we covered.

Computers can do amazing things but they need to be told what to do. We tell them what to do by running software. Software is generally built from *source code*, which is readable by humans, and is a sequence of basic steps that a computer will follow exactly. There are many different software languages (such as JavaScript), with different ways of expressing these basic steps, however most of the main concepts are universal. Our job is to break down problems into these steps, and then the computer will *run* that code when told. Generally, when source code is run, it executes from the first line linearly to the last line. However, we can change this flow by adding “fork-in-the-road” (conditional) or “do-that-part-a-few-times” (loop) structure to our source code.

A variable is a labeled, local space that can contain a value. We refer to a variable by its label if we want to read or change that value. Values can be a few different types, such as *numbers* or *strings* or *booleans*. A string is a sequence of characters, and a boolean is simply a true/false value. JavaScript (also known as JS) automatically changes values from one type to another as needed.

A single-equals (=) is used to set values in variables, and can be combined with normal mathematical operators (+ - \* /). The == operator compares two values, allowing JS to convert data types if needed; the === operator does *not* allow the types to be converted. We print to the developer console using the `console.log` function, which accepts a string (or converts inputs to a string).

We can examine the values of variables, and divert the flow of source code execution depending on those values, using the `IF` statement. This can be combined with an `ELSE`, to cover the other side of a conditional as well; these `IF...ELSE` statements can be nested. One can create compound comparisons using logical operators that represent *and*, *or* and *not* (&&, ||, !).

To execute a specific piece of source code multiple times, there are two different types of *loops* available. A `FOR` loop is particularly useful when you know exactly how many times you need to loop; in other cases, a `WHILE` loop is simple and flexible. With both kinds of loops, you can use `BREAK` and `CONTINUE` statements to change the flow of code (to exit the loop or skip a certain iteration).

We can extract a piece of source code into a `FUNCTION` so that it can easily be called repeatedly. Functions (like variables) have labels, and to call a function, we list the function name with () following it. A function can require one or more values from outside, and we pass those values in by using parameters, which are included between the parentheses when calling the function, and similarly specified between the parentheses when defining the function.

# Chapter 1 – Fundamentals



OK Ninjas-in-training, use your new knowledge. Can you solve these?

## Setting and Swapping

Set `myNumber` to 42. Set `myName` to your name. Now swap `myNumber` into `myName` & vice versa.

## Print -52 to 1066

Print integers from -52 to 1066 using a `FOR` loop.

## Don't Worry, Be Happy

Create `beCheerful()`. Within it, `console.log` string "good morning!" Call it 98 times.

## Multiples of Three – but Not All

Using `FOR`, print multiples of 3 from -300 to 0. Skip -3 and -6.

## Printing Integers with While

Print integers from 2000 to 5280, using a `WHILE`.

## You Say It's Your Birthday

If 2 given numbers represent your birth month and day *in either order*, log "How did you know?", else log "Just another day...."

## Leap Year

Write a function that determines whether a given `year` is a leap year. If a year is divisible by four, it is a leap year, unless it is divisible by 100. However, if it is divisible by 400, then it is.

## The Final Countdown

This is based on "Flexible Countdown". The parameter names are not as helpful, but the problem is essentially identical; don't be thrown off! Given 4 parameters (`param1, param2, param3, param4`), print the multiples of `param1`, starting at `param2` and extending to `param3`. One exception: if a multiple is equal to `param4`, then skip (don't print) that one. Do this using a `WHILE`. Given `(3, 5, 17, 9)`, print `6, 12, 15` (which are all of the multiples of `3` between `5` and `17`, except for the value `9`).

## Print and Count

Print all integer multiples of 5, from 512 to 4096. Afterward, also log how many there were.

## Multiples of Six

Print multiples of 6 up to 60,000, using a `WHILE`.

## Counting, the Dojo Way

Print integers 1 to 100. If divisible by 5, print "Coding" instead. If by 10, also print " Dojo".

## What Do You Know?

Your function will be given an input parameter `incoming`. Please `console.log` this value.

## Whoa, That Sucker's Huge...

Add odd integers from -300,000 to 300,000, and `console.log` the final sum. Is there a shortcut?

## Countdown By Fours

Log positive numbers starting at 2016, counting down by fours (exclude 0), *without* a `FOR` loop.

## Flexible Countdown

Based on earlier "Countdown By Fours", given `lowNum, highNum, mult`, print multiples of `mult` from `highNum` down to `lowNum`, using a `FOR`. For `(2, 9, 3)`, print `9 6 3` (on successive lines).

# Chapter 1 – Fundamentals

---

## Return Values

Parameters give functions a lot more flexibility. However, sometimes you don't want a function to do *all* the work; maybe you just want it to give you information so that then your code can do something based on the answer it gives you. This is when you would use the *return value* for a function.

Functions have names (usually). They (often) have parameters. They have code that will run when the function is executed. They generally have a *return value* as well, which is simply a value that is returned to the caller when the function finishes executing. Not all functions have *return* values, and looking at source code you might think that not all functions have a return statement. However, they indeed *do*, because if there is nothing stated, an implicit `return` is added automatically at the end of the function.

In JavaScript, if a caller “listens” to a function that ends with `return`, the caller receives `undefined`. If we want to be more helpful, we can explicitly return a value (for example, a variable or a literal). In other words, our function could `return myNewName;` or could `return "Zaphod";`. In either case, once the `return` statement runs, any subsequent lines of code in our function will not be executed. When program execution encounters a `return`, it exits the current function immediately.

If functions can return values, to *tell us the answer*, then whoever calls those functions must *listen for that answer*. It is easy enough to execute a function (`greetSomeone(nameStr)`, below left) that has no return. If a function does return a value, then to actually *receive* that value the caller should save the result into a var, or otherwise “listen” to what it says (`tellMeAGoodJoke()`, below right):

```
// Calling a function that          // This one DOES return a value
// does NOT return a value        var joke = tellMeAGoodJoke();
greetSomeone("Claire");           console.log(joke);
```

In the above, `tellMeAGoodJoke()` presumably returns a string, which we copy into local variable `joke` and display. See below for how to declare that function, but beware the function's sequel!

```
function tellMeAGoodJoke() {
  var jokeStr = "Have you heard about corduroy pillowcases?";
  jokeStr = jokeStr + " .... They're making headlines!";
  return jokeStr;
  jokeStr += "Thanks, I'm here all week...";    // never runs!
}

// Maybe it's a good joke, but it's a BAD FUNCTION. You can't return twice!
function tellMeAnotherOne() {
  var aJoke = "How many surrealists does it take to screw in a lightbulb?";
  return aJoke;
  return " .... A fish.";      // Wha? Oh I get it...but JavaScript won't.
}                                // Remember: you can't return twice!
```

# Chapter 1 – Fundamentals

---

## Arrays

An array is like a cabinet with multiple drawers, where each drawer stores a number, string, or even another array. In JavaScript, arrays are created by code like this:

```
var arr = [2, 4, 6, 8];      // create array with four distinct values
```

In our example, we created an array called `arr`. This array `arr` is like a file cabinet with three drawers. To look into one of these file cabinets, we have to specify which one. Each drawer is numbered, starting at the number 0 (not 1). The first drawer, drawer 0, has a value of 2; the second drawer, labeled 1, has a value of 4; the next drawer, which we call drawer 2, has a value of 6. In our code, we reference the different locations in an array by specifying the 'drawer number', which is really the offset from the beginning of our array. Specifically, we read an array value by putting its offset between square-brackets, as follows:

```
console.log(arr[1]);        // "4" (Not 2 - this is at arr[0])
```

Arrays have three important built-in properties: `push`, `pop` and `length`. We add a value to the end of our array (which lengthens it by one) with `push`:

```
arr.push(777);            // arr was [2,4,6,8], is now [2,4,6,8,777]
```

This pushes a new value *onto the end of the array*, so `arr` has a new value and is slightly longer – it is now `[2,4,6,8,777]`.

Similarly, we *remove (and return) the value at the end of the array* (and we shorten our array by one) by using the `pop` function:

```
var last = arr.pop();      // arr was [2,4,6,8,777], is now [2,4,6,8]
console.log(last);         // "777" - this is what pop() returned
```

The examples we've used above have lengthened and shortened our array. We see this on the page by just looking at all the values, but how would we quickly do this in code? We would use a useful property on every array called `length`. This is attached to each array like `pop` and `push` are, but it is not a function, so you do not need parentheses when using it:

```
console.log(arr);          // "[2,4,6,8]"
console.log(arr.length);    // "4" - vals are stored at indices 0,1,2,3
```

Said another way, `arr.length` is always one greater than `arr`'s highest populated index.

# Chapter 1 – Fundamentals

---

## Writing Values into Arrays

In the previous example, our array `arr` had four (sometimes five) values. Each value in an array has its own space set aside for it, like the different drawers in a file cabinet.

```
var arr = [2, 4, 6, 8];      // create array with four distinct values
```

The beginning drawer (at index 0) has a value of 2; the second drawer (index 1) has a value of 4; the third drawer (index 2) has a value of 6. We change values within an array in the same way that we reference its values when reading from it: we enclose the index in square brackets, like this:

```
arr[1] = 10;                // arr was [2, 4, 6, 8], is [2, 10, 6, 8].
```

This statement sets `arr[1]` to be 10. It puts a value of 10 into `arr[1]`, the `arr` cabinet at index 1.

We often need to swap the values of two variables (this will be handy later, for algorithms such as “reverse an array”). We can treat the spaces in an array exactly the same. What if we tried swapping the value at index 1 with the value at index 3? We might try something like the below:

```
// arr is currently [2,10,6,8]. We want to change it to [2,8,6,10]
x[1] = x[3];
x[3] = x[1];
console.log(x);           // ...but this code won't work quite right.
                           // arr got messed up! It is [2,8,6,8].
```

The code above wouldn’t quite work. For example, let’s talk through this code step by step.

- Before starting, `arr` is equal to `[2, 10, 6, 8]`.
- In line 2, we set `arr[1]` to be the value in `arr[3]`, which is 8. Therefore, `arr` becomes `[2, 8, 6, 8]`.
- When we run line 3, we set `arr[3]` to be the value in `arr[1]`: 8. Thus, we overwrite an 8 with an 8, and `arr` remains `[2, 8, 6, 8]`.

We can avoid this problem by creating a temporary variable to store the value of `arr[1]` before it is overwritten. To swap values in the array (or elsewhere), always use a temporary variable. For example:

```
arr = [2, 10, 6, 8];
temp = arr[1];              // arr == [2, 10, 6, 8], temp == 10
arr[1] = arr[3];            // arr == [2, 8, 6, 8], temp == 10
arr[3] = temp;              // arr == [2, 8, 6, 10], temp == 10
console.log(arr);           // displays [2,8,6,10]
```

Success! Now, onward to algorithm challenges that use arrays.

# Chapter 1 – Fundamentals

---

## Countdown

Create a function that accepts a number as an input. Return a new array that counts down by one, from the number (as array's 'zero'th element) down to 0 (as the last element). How long is this array?

## Print and Return

Your function will receive an array with two numbers. Print the first value, and return the second.

## First Plus Length

Given an array, return the sum of the first value in the array, plus the array's length. What happens if the array's first value is *not* a number, but a string (like "what?") or a boolean (like `false`)?

## Values Greater than Second

For `[1, 3, 5, 7, 9, 13]`, print values that are greater than its 2<sup>nd</sup> value. Return how many values this is.

## Values Greater than Second, Generalized

Write a function that accepts *any* array, and returns a new array with the array values that are greater than its 2<sup>nd</sup> value. Print how many values this is. What will you do if the array is only one element long?

## This Length, That Value

Given two numbers, return array of length `num1` with each value `num2`. Print "Jinx!" if they are same.

## Fit the First Value

Your function should accept an array. If value at `[0]` is greater than array's length, print "Too big!"; if value at `[0]` is less than array's length, print "Too small!"; otherwise print "Just right!".

## Fahrenheit to Celsius

Kelvin wants to convert between temperature scales. Create `fahrenheitToCelsius(fDegrees)` that accepts a number of degrees in Fahrenheit, and returns the equivalent temperature as expressed in Celsius degrees. For review, `Fahrenheit = (9/5 * Celsius) + 32`.

## Celsius to Fahrenheit

Create `celsiusToFahrenheit(cDegrees)` that accepts number of degrees Celsius, and returns the equivalent temperature expressed in Fahrenheit degrees.

(optional) Do Fahrenheit and Celsius values equate at a certain number? Scientific calculation can be complex, so for this challenge just try a series of Celsius integer values starting at 200, going downward (descending), checking whether it is equal to the corresponding Fahrenheit value.

# Chapter 1 – Fundamentals

---

## Combining Arrays and FOR Loops

In programming, it's very common to loop through each array value. We can do this as follows:

```
var nums = [1,3,5,7];                      // set up our loop
for (var idx = 0;idx < nums.length;idx++)    // for each index in arr...
{
  console.log(nums[idx]);                  // ...print the value
}
```

This prints each value in the array, using a **FOR** loop that iterates once for each array value.

What if we wanted an array with multiples of 3 up to 99,999? We accomplish this with the code below:

```
var arr = [];                                // create empty array
for (var val = 3;val <= 99999;val += 3)      // val will be 3,6,...99999
{
  arr.push(val);                            // add each val to arr
}
console.log(arr);                           // [3,6,9,12,..., 99999]
```

You will frequently write **FOR** loops that, at the end of each loop iteration, compare a variable (like `idx`) to your `Array.length`. Remember that when you call `push()` or `pop()`, your array's `.length` does change. Copy this value into a local variable, if you need its original value.

Here's an example of code that does not work as the programmer intended:

```
// BADCODE - intentionally buggy
function addEvenCount(arr) {
  // Count array even values & add that number to end of array
  for (var idx = 0; idx < arr.length; idx++) {
    if (idx == 0) {
      arr.push(0);                          // First time, add 0 to end.
    }
    if (arr[idx] % 2 == 0) {                // Then just add to it as we go.
      arr[arr.length - 1] += 1;
    }
  }
}                                         // Oops! We counted "2" as well.
```

The problem, of course, is that if we push our zero to the end of the array, then `arr.length` increments, and now our **FOR** loop will run on the index we just added as well. Given `[0,3,6,5]`, we want the array to be changed to `[0,3,6,5,2]`, but it would instead change to `[0,3,6,5,3]`.

# Chapter 1 – Fundamentals

---

## Biggie Size

Given an array, write a function that changes all positive numbers in the array to “big”. Example: `makeItBig([-1, 3, 5, -5])` returns that same array, changed to `[-1, "big", "big", -5]`.

## Print Low, Return High

Create a function that takes array of numbers. The function should print the lowest value in the array, and *return* the highest value in the array.

## Print One, Return Another

Build a function that takes array of numbers. The function should *print* second-to-last value in the array, and *return* first odd value in the array.

## Double Vision

Given array, create a function to return a *new* array where each value in the original has been doubled. Calling `double([1, 2, 3])` should return `[2, 4, 6]` without changing original.

## Count Positives

Given array of numbers, create function to replace last value with number of positive values. Example, `countPositives([-1, 1, 1, 1])` changes array to `[-1, 1, 1, 3]` and returns it.

## Evens and Odds

Create a function that accepts an array. Every time that array has three odd values in a row, print “**T**hat’s **o**dd!” Every time the array has three evens in a row, print “**E**ven **m**ore **s**o!”

## Increment the Seconds

Given an array of numbers `arr`, add 1 to every second element, specifically those whose index is odd (`arr[1], [3], [5]`, etc). Afterward, `console.log` each array value and return `arr`.

## Previous Lengths

You are passed an array containing strings. Working within that same array, replace each string with a number – the length of the string at *previous* array index – and return the array.

## Add Seven to Most

Build function that accepts array. Return a new array with all values *except first*, adding 7 to each. Do not alter the original array.

## Reverse Array

Given array, write a function that reverses values, in-place. Example: `reverse([3, 1, 6, 4, 2])` returns same array, containing `[2, 4, 6, 1, 3]`.

## Outlook: Negative

Given an array, create and return a new one containing all the values of the provided array, made negative (*not simply multiplied by -1*). Given `[1, -3, 5]`, return `[-1, -3, -5]`.

## Always Hungry

Create a function that accepts an array, and prints “**y**ummy” each time one of the values is equal to “**f**ood”. If no array elements are “**f**ood”, then print “**I’m h**ungry” once.

## Swap Toward the Center

Given array, swap first and last, third and third-to-last, etc. Input `[true, 42, "Ada", 2, "pizza"]` becomes `["pizza", 42, "Ada", 2, true]`. Change `[1, 2, 3, 4, 5, 6]` to `[6, 2, 4, 3, 5, 1]`.

## Scale the Array

Given an array `arr` and a number `num`, multiply all values in `arr` by `num`, and return the changed array `arr`.

# Chapter 1 – Fundamentals

## Using a T-Diagram

When trying to decipher complex code, particularly if someone else wrote it, a T-diagram can prove very valuable. Eventually you may not need them, but while you are early in your journey toward becoming a self-sufficient developer, you should definitely use them frequently. Here's how they work:

A T-diagram is a way to record the state of all your local variables, including arrays and their indices. After every assignment in your code, update the diagram. Before each conditional (**IF/ELSE**, or each time through a **WHILE** or **FOR** loop) you can check the variable's value in your T-diagram to predict how the code will behave. Let's walk through a short function.

Here's the code we will trace through. No sweat, right?

```
1. var arr = [1,3,5];
2. var idx = arr[1];
3. arr[idx] = arr[0];
4. idx--;
5. arr.push(arr[idx]);
6. idx = arr.length - arr[idx];
7. console.log(idx + arr[idx]);
```

Lines 1-2: following these, your T-diagram looks like this. With the diagram we see that we are changing `idx` to become 3.

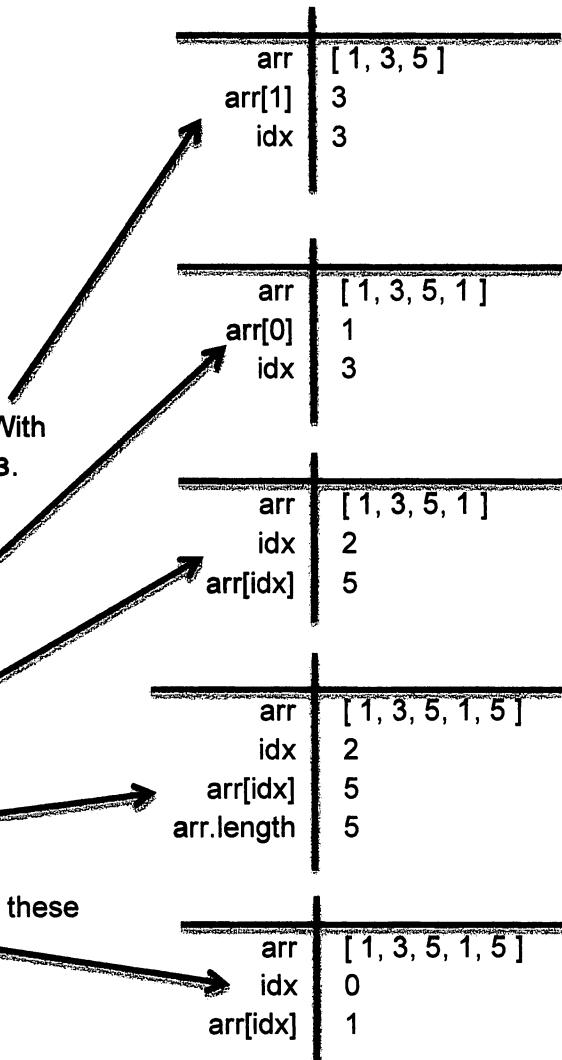
Line 3: the diagram tells you that you are changing `arr[3]` to become 1, so your T-diagram becomes this:

Line 4: after decrementing `idx`, we have this diagram, including `arr[idx]` we want for the next line of code:

Line 5: we easily see that we are pushing 5 to array's end. Now diagram is this (including `arr.length`):

Line 6: we set `idx` to new value. With the diagram we see these values (5 - 5, or 0). We can update `arr[idx]` as well:

Line 7: finally, we can make the simple sum (0 + 1), to determine what will be printed by `console.log: 1`.



We hope this quick walkthrough shows how T-diagrams bring clarity even to fairly complicated code.

# Chapter 1 – Fundamentals

---

## Only Keep the Last Few

Stan learned something today: that reducing an array's `.length` immediately shortens it by that amount. Given array `arr` and number `x`, remove all except the last `x` elements, and return `arr` (changed and shorter). Given `([2, 4, 6, 8, 10], 3)`, change the given array to `[6, 8, 10]` and return it.

## Math Help

Cartman doesn't really like math class and needs help. You are given two numbers – the coefficients `M` and `B` in the equation  $Y = MX + B$ . Build a function that returns the X-intercept (Cartman's older cousin Charlie wisely reminds him that X-intercept is the value of X where Y equals zero, but he just scoffs).

## Poor Kenny

Kenny tries to stay safe, but somehow everyday something happens. If there is a 10% chance of volcano, 15% chance of tsunami, 20% chance of earthquake, 25% chance of blizzard, and 30% chance of meteor strike, write function `whatHappensToday()` to print the outcome.

## What Really Happened?

Kyle (smarter than Kenny) notes that the chance of one disaster is totally unrelated to the chance of another. Change `whatHappensToday()` function to create `whatReallyHappensToday()`. In this new function *test for each disaster independently*, instead of assuming exactly one disaster will happen. In other words, with this new function, *all five* might occur today – or *none*. Maybe Kenny will survive!

## Soaring IQ

Your time with us will definitely make you smarter! Let's say a new Dojo student, Bogdan, entered with a modest IQ of 101. How smart would Bogdan be at the end of the bootcamp, if his IQ rose by .01 on the first day, then went up by an additional .02 on the second day, up by .03 more on the third day, etc.... all the way until increasing by .98 on his 98<sup>th</sup> day (the end of 14 full weeks)?

## Letter Grade

Mr. Cerise teaches high school math. Write a function that assigns and prints a letter grade, given an integer representing a score from 0 to 100? Those getting 90+ get an 'A', 80-89 earn 'B', 70-79 is a 'C', 60-69 should get a 'D', and lower than 60 receive 'F'. For example, given 88, you should log "Score: 88. Grade: B". Given the score 61, log the string "Score: 61. Grade: D".

## More Accurate Grades

For an additional challenge, add '-' signs to scores in the bottom two percent of A, B, C and D scores, and "+" signs to the top two percent of B, C and D scores (sorry, Mr. Cerise never gives an A+). Given 88, `console.log "Score: 88. Grade: B+"`. Given 61, log "Score: 61. Grade: D-".

# Chapter 1 – Fundamentals

---

## Comments

Source code containing good comments is a joy to work with. You don't spend as much time trying to figure it out, because the creator cared enough to spend just a few moments ahead of time to explain it. There are two ways to write comments in JavaScript source code.

One option is `//`. After a double-slash, the *rest of that line* is a comment. Your code might look like this:

```
// This is a very friendly function, if I do say so myself.
function greetSomeone(person) {
    if (person == "Martin")          // Check whether it is Martin...
    {
        console.log("Yo dawg, howz it goin?");
    }
    else                            // if not, probably some normal human.
    {
        console.log("Greetings Earthling!");
    }
}
```

Another option is `/* ... */`. These `/*` and `*/` bookends can span multiple lines, and *everything between them* is considered a non-source-code comment. This style would look like this:

```
/*
    Simple function that responds directly if the person is Martin,
    otherwise it provides a more generic salutation. No return value.
*/
function greetSomeone(person)
{
    if (person == "Martin") {
        console.log("Yo dawg, howz it goin?");
    } else {
        console.log("Greetings Earthling!"); /* no clue who it is... */
    }
}
```

Both are commonly used. Quick comments sprinkled throughout your course are usually `//`; larger comment blocks are often `/* ... */`. The main thing is simply to add a few comments. The next person to work with the code (perhaps a future YOU when you have forgotten details) will appreciate it.

Now that you have been introduced to the foundation concepts of source code execution, variables, conditionals, loops, arrays, functions, parameters, return values and comments, you are ready to continue onward to the rest of the algorithm materials. Enjoy!

# Chapter 1 – Fundamentals

---

## Short Answer Questions: Fundamentals

What is source code?

What makes computers so “smart”, anyway?

What is the purpose of a programming language?

What are 3 examples of programming languages? Why are there so many of these?

What is a variable? Why are variables useful?

What is the difference between a single-equals (=) and a double-equals (==)?

What is the difference between a double-equals (==) and a triple-equals (===)?

Why does the developer console exist?

When we talk about “conditional” statements, what does that mean? What is an example?

Why would we want FOR or WHILE loops in our source code?

When would you use a WHILE loop, instead of a FOR loop?

What is a function? Why would we use functions?

How many values can you receive back from a function? How many values can you send in?

What is an array? How many values does it hold?

What is a T-diagram and why should I know how to use one?

What are the two ways to comment JS code? When would you use one versus the other?

## Weekend Challenge: Fundamentals

This weekend, for a challenge, create a *fill-in-the-blank* quiz game. Ask the user’s name, then refer to the user by name as you ask him/her a series of questions that you have stored in an array. Use the `prompt()` function to get each input from the user and compare it to the answer you expected. When the user enters “Q” (for quit), or perhaps when the user hits [Cancel], exit the game and print the statistics of the game to the console: user name, number of questions answered and questions correct.

## Fundamentals Review

This chapter covered a number of very important topics. Most importantly, we introduced you to the creation, reading, and changing of arrays, including using arrays in conjunction with FOR loops. We also showed how functions can not only accept input values (parameters), but also output a value back to the caller as well (return values). We gave our first example of how to use a T-diagram (there will be more of these), to make sense of a piece of source code. Finally, we demonstrated two ways to add comments to your source code, after talking about the importance of good commenting.

## The “Basic 13”

These are Coding Dojo's foundation “Basic 13” algorithm challenges.

Can you finish these in less than two minutes each?

### Print 1-255

Print all the integers from 1 to 255.

### Print Ints and Sum 0-255

Print integers from 0 to 255, and with each integer print the sum so far.

### Find and Print Max

Given an array, find and print its largest element.

### Array with Odds

Create an array with all the odd integers between 1 and 255 (inclusive).

### Greater than Y

Given an array and a value Y, count and print the number of array values greater than Y.

### Max, Min, Average

Given an array, print the max, min and average values for that array.

### Swap String For Array Negative Values

Given an array of numbers, replace any negative values with the string 'Dojo'.

### Print Odds 1-255

Print all odd integers from 1 to 255.

### Iterate and Print Array

Iterate through a given array, printing each value.

### Get and Print Average

Analyze an array's values and print the average.

### Square the Values

Square each value in a given array, returning that same array with changed values.

### Zero Out Negative Numbers

Return the given array, after setting any negative values to zero.

### Shift Array Values

Given an array, move all values forward by one index, dropping the first and leaving a '0' value at the end.

This chapter, we review basic blocks of programming: conditionals, logic operators, loops and a few techniques. All of the following concepts are used in this chapter:

**variables      functions      for loops      while loops      conditional (if / else) statements**  
**console.log    parameters    return values    Math.random | Math.ceil | Math.floor | Math.trunc**

Review: define **variable**. Think of it as simply an empty container with a label. Once you put a value into the container, you can refer to this value by the label. We put a value into a variable using single-equals, which you read as "is set to a value of". In other words, `var name = "Zaphod"` can be read as *Variable labeled "name" is set to a value of "Zaphod"*. After this line of code, when you refer to `"name"`, you get a value of `"Zaphod"`. If you are still getting used to the idea of variables, *don't panic*.

Sigma

Implement function `sigma(num)` that given a number, returns the sum of all positive integers up to number (inclusive). Ex.: `sigma(3) = 6` (or `1 + 2 + 3`); `sigma(5) = 15` (or `1 + 2 + 3 + 4 + 5`).

## Factorial

Just the Facts, ma'am. Factorials, that is. Write a function `factorial(num)` that, given a number, returns the product (multiplication) of all positive integers from 1 up to number (inclusive). For example, `factorial(3) = 6` (or `1 * 2 * 3`); `factorial(5) = 120` (or `1 * 2 * 3 * 4 * 5`).

## Star Art

Assume that you have a text field that is exactly 75 characters long. You want to fill it with spaces and asterisks ('\*'), sometimes called *stars*. You should print the given number of asterisks consecutively. Depending on which function is called, those stars should be *left-justified* (first star would be very first char in the text field), or *right-justified* (last star would be very last char in the text field, with potentially some number of spaces at beginning of text field before the block of stars start), or *centered* in the 75-character text field (with same number of spaces on either side of the block of stars, plus/minus one).

- Write a function `drawLeftStars (num)` that accepts a number and prints that many asterisks.
  - Write a function `drawRightStars (num)` that prints 75 characters total. Stars should build from right side. The last `num` characters should be asterisks; the other 75 should be spaces.
  - Write function `drawCenteredStars (num)` that prints 75 characters total. The stars should be centered in the 75. The middle `num` characters should be asterisks; the rest of the 75 spaces.
  - (optional) Create epic text-art Empire vs. Rebellion battle, using formations like `(=*=)` and `><`.

## Character Art

**From the above, derive the following that accept and draw the given characters, not just asterisks:**

- `drawLeftChars(num, char)` // for all three of these, you
  - `drawRightChars(num, char)` // can safely assume that 'char'
  - `drawCenteredChars(num, char)` // is a string with length 1

## Chapter 2 – Fundamentals, Part II

---

It is imperative at this point in the bootcamp that you can rapidly complete the mandatory coding challenges from the **Algorithm Platform**. If you have not yet correctly answered each of them in under two minutes, then revisit the Algorithm Platform, “Reset All Challenges”, and see how speedily you can complete them. Repeat until you can reliably finish each of them in less than two minutes.

### **Modulo Operator**

So far you have learned about basic arithmetic operators to *add* (+), *subtract* (-), *multiply* (\*) and *divide* (/). You may also have realized that JavaScript uses the + operator to *concatenate strings* as well!

Now we want to introduce you to another operator, called *modulo* (%). Modulo is a companion operator to divide – think of it as “remainder”. Given two numbers, modulo divides the second number into the first number an integer number of times, and returns the remainder. Examples: (34 % 6) is 4 because 34 integer-divides into 6 five times (30), leaving a remainder of 4. Modulo is great for determining if a number is even/odd: (16 % 2) is 0: it is even. Is 42 a multiple of ten? (42 % 10) is 2: no it is not.

#### **Threes and Fives**

Create **threesFives()** that adds values from 100 and 4000000 (inclusive) *if that value is evenly divisible by 3 or 5 but not both*. Display the final sum in the console.

**Second:** Create **betterThreesFives(start, end)** that allows you to enter arbitrary start and end values for your range. Think of **threesFives()** as **betterThreesFives(100, 4000000)**.

#### **Generate Coin Change**

Change is inevitable (especially when breaking a twenty). Make **generateCoinChange(cents)**.

Accept a number of American cents, compute and print how to represent that amount with smallest number of coins. Common American coins are pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents).

**Second:** can you simplify/shorten your code?

Example output, given (94):

```
94 cents can be represented by:  
quarters: 3  
dimes: 1  
nickels: 1  
pennies: 4
```

**Third:** add half-dollar (50 cents) and dollar (100 cents) coins with 40 additional characters or less.

#### **Messy Math Mashup**

Create a function **messyMath(num)** that will return the following sum: add all integers from 0 up to the given **num**, except for the following special cases of our **count** value:

- 1 . If current **count** (not **num**) is evenly divisible by 3, don't add to **sum**; skip to the next **count**;
- 2 . Otherwise, if current **count** is evenly divisible by 7, include it *twice* in **sum** instead of once;
- 3 . Regardless of the above, if current **count** is exactly 1/3 of **num**, return -1 immediately.

For example, if given num is 4, return 7. If given num is 8, return 34. If given num is 15, return -1.

## Chapter 2 – Fundamentals, Part II

---

### Twelve-Bar Blues

Write a function that `console.logs` the number 1, then "chick", then "boom", then "chick", then 2, then "chick", "boom", "chick" – continuing the same cycle for each number up to (including) 12.

### Fibonacci

Create a function to generate *Fibonacci* numbers. In this famous mathematical sequence, each number is the sum of the previous two, starting with values 0 and 1. Your function should accept one argument, an index into the sequence (where 0 corresponds to the initial value, 4 corresponds to the value four later, etc). Examples: `fibonacci(0) = 0` (given), `fibonacci(1) = 1` (given), `fibonacci(2) = 1` (`fib(0)+fib(1)`, or `0+1`), `fibonacci(3) = 2` (`fib(1)+fib(2)`, or `1+1`), `fibonacci(4) = 3` (`1+2`), `fibonacci(5) = 5` (`2+3`), `fibonacci(6) = 8` (`3+5`), `fibonacci(7) = 13` (`5+8`), etc.

### Sum to One Digit

Kaitlin sees beauty in numbers, but also believes that less is more. Implement `sumToOne(num)` that sums a given integer's digits repeatedly until the sum is only one digit. Return that one-digit result. Example: `sumToOne(928)` returns 1, because  $9+2+8 = 19$ , then  $1+9 = 10$ , then  $1+0 = 1$ .

### Clock Hand Angles

Regardless of how hard a Dojo student works (and they *should* work hard), they need time now and then to unwind – like hands on a clock. Traditional clocks are increasingly uncommon, but most can still read an analog clock's hands of hours, minutes and seconds. Create `clockHandAngles(seconds)` that, given a number of seconds since 12:00:00, prints angles (in degrees) of the hour, minute and second hands. As review, 360 degrees form a full rotation. For input of 3600 secs (equivalent to 1:00:00), print `"Hour hand: 30 degs. Minute hand: 0 degs. Second hand: 0 degs."` For an input parameter `seconds` of 119730 (which is equivalent to 9:15:30 plus 24 hours!), you should log `"Hour hand: 277.745 degs. Minute hand: 93 degs. Second hand: 180 degs."` Note: in the second example, the angle for the minute hand is not simply 90 degrees; it has advanced a bit further, because of the additional 30 seconds in that minute so far.

**Second:** also calculate and print degrees for an additional "week hand" that rotates once each week.

### Is Prime

Return whether a given integer is prime. Prime numbers are only evenly divisible by themselves and 1. Many highly optimized solutions exist, but for now just create one that is *easy to understand and debug*.

## Chapter 2 – Fundamentals, Part II

---

Being able to write a T-diagram to keep track of your variables while you write out an algorithm by hand is extremely beneficial. You should use a T-diagram for algorithm challenges this chapter.

### Math Library

A library is a related set of functions and values that have been grouped together under a common name. Traditionally this is done for less common functions, so they can be excluded from certain minimized versions of a language (e.g.: if we want a micro-JavaScript for some future Apple Ring™). Looking back now, it seems incredible that a language without math functions would be useful. Nonetheless JavaScript has grouped certain numerical functions and values into the Math library. When using these, put `Math.` before it (just like when using `log()` from `console` library!). Note that libraries are not limited to *just* functions. They can also include values such as `Math.PI`. Right now, however, we will focus on four functions in this library: `random()`, `floor()`, `ceil()` and `trunc()`.

The first, `Math.random`, returns a randomly generated decimal number between 0 and 1. It can theoretically return zero, but it cannot return one; for this reason you can think of it as returning *some number between zero and ‘almost-one’*. The other three functions are related: they accept a decimal number and return an *integer*. Given an integer, all three leave it *unchanged*. Otherwise, floor is a pessimist, ceil is an optimist, and trunc is a simplifier. `Math.floor` makes negative numbers more negative, and positives less positive. Conversely, `Math.ceil` makes positives more positive, and negatives less negative. `Math.trunc` drops any fraction, moving the number toward zero.

`Math.floor(2.718)` and `Math.trunc(2.718)` both return 2, but `Math.ceil(2.718)` returns 3. `Math.floor(-3.1416)` is -4; both `Math.trunc(-3.1416)` and `Math.ceil(-3.1416)` are -3.

Naturally, `Math.ceil(42) == Math.trunc(42) == Math.floor(42) == 42`.

One last idea. What if you want a random integer as low as 51 and as high as 100? `Math.random()` is “from 0 to almost-one”. `Math.random() * 50`, then, is “from 0 to almost-50”. Let’s turn those decimal ranges into integers: `Math.trunc(Math.random() * 50)` is “50 possible integers from 0 to 49”. Let’s add an offset, so we start at 51: `Math.trunc(Math.random() * 50) + 51` is perfect. Whew!

### Rockin’ the Dojo Sweatshirt

Ever since you arrived at the Dojo, you wanted one of those cool Coding Dojo sweatshirts – maybe even more than one. Let’s say they cost \$20 (including tax), but friendly Josh gives a 9% discount if you buy two, a nice 19% discount if you buy three, or a sweet 35% discount if you buy four or more. He only accepts cash and says he doesn’t have coins, so you should round up to the nearest dollar. Build function `sweatshirtPricing(num)` that, given a number of sweatshirts, returns the cost.

### Clock Hand Angles, Revisited

Return to your previous `clockHandAngles` solution. Allow fractional values for input `seconds`, but change your implementation to print only *integer* values for angles (in degrees) of the various hands.

## Chapter 2 – Fundamentals, Part II

---

### Using Modulo to Extract a Digit

If variable `myBigNum` contained a big number, how would you get the value of the ‘hundreds’ digit? (To review, the *hundreds* digit of 32768 is ‘7’, since it is thirty-two thousand, seven hundred sixty-eight.)

First, you might `myBigNum = myBigNum / 100`, to shift the decimal point to exactly where you want (result: 327.68). Then, `Math.floor(myBigNum)` could remove any decimal leftovers (result: 327).

Finally, you could use `%` to extract only the ‘ones’ digit. *What is % and how does it work?*

The `%` (modulo) operator often goes with `/` (divide). Basically, `% divides first number evenly by second, and returns remainder`. Example: `255 % 20` is `15`, and `16 % 2` is `0`. And ... `327 % 10` becomes the `7` we want. Putting it all together, we print the ‘hundreds’ digit like this:

```
console.log(Math.floor(myBigNum / 100) % 10);
```

Let’s decode this. Formulas are computed inside out; so start with the division inside the parentheses. Once we’ve done that, our formula is simplified:

```
console.log(Math.floor(327.68) % 10);
```

Then comes the floor call. After evaluating that, our formula is:

```
console.log(327 % 10);
```

And finally comes the modulo operation, after which we have this!

```
console.log(7);
```

### Extract-o-matic

Create the `extractDigit(num, digitNum)` function that given a number and a digit number, returns the numeral value of that digit. 0 represents the ones digit, 1 represents the tens digit, etc. Given `(1824, 2)`, return `8`. Given `(1824, 0)`, return `4`. Given `(1824, 7)`, return `0`.

**Second:** handle negative `digitNum` values, where `-1` represents tenths digit (`0.x`), `-2` represents hundredths digit (`0.0x`), etc. Given `(123.45, -1)`, return `4`.

**Third:** handle negative `num` values as well, doing what you think is appropriate.

### Most Significant Digit

If you already know who Ada Lovelace is, that’s great! In a History of Science, she is *significant*. Given number of any size, return the most *significant* digit. If you already know what strings are, that’s great! However, don’t use them here ☺. Hint: use `WHILE` to bring the most significant digit into range where you can use the friendly modulus operator (%). The most significant digit is the *leftmost non-zero digit* of a number. Given `12345`, return `1`. Given `67.89`, return `6`. Given `0.00987`, return `9`.

**Second:** handle negative `num` values as well, doing what you think is appropriate.

# Chapter 2 – Fundamentals, Part II

---

## Variables that Live Longer than a Single Function Call

If you declare a `var` within a function, it is created when entering, and destroyed when exiting that function. For a `var` to stay alive after you leave, you must declare it outside. That declaration will be called only once, when the file is loaded – including any initialization done on that variable. This can be useful if you want functions to “remember” values between successive calls to them. You should contain variables within a function when possible, but you can declare them outside if needed.

## Gaming Fun(damentals)

It's New Year's Eve, so let's play some dice games! It'll be fun. What could go wrong?

- 1) Create function `rollOne()` to return a randomly selected `integer` between 1 and 6 (inclusive).
- 2) Second, create a function `playFives(num)`, which should call `rollOne()` multiple times – '`num`' times, in fact, where '`num`' is input parameter to `playFives(num)`. Each time, it should print the value `rollOne()` returns, and if that return value is 5, also print “That's good luck!”
- 3) Third, create a new function named `playStatistics()`, which should call `rollOne()` eight times (but not print anything after each call). After the last of these eight calls, it should print out the lowest and highest values that it received from `rollOne`, among those eight calls.
- 4) Fourth, make a copy of `playStatistics` and add code to make `playStatistics2()`, so that at the end (in addition to printing high/low rolls), it also prints the total sum of all eight rolls.
- 5) Fifth, copy `playStatistics2` and add code to it to make `playStatistics3(num)`, so that it will roll as many times as you want, instead of always doing this eight times.
- 6) Finally, make a copy of `playStatistics3` and change it to create `playStatistics4(num)`, so that at the end instead of the total sum, it prints the average roll.

## Statistics Until Doubles

Here's another game for our New Year's Eve party. Implement a '20-sided die' that randomly returns integers between 1 and 20 inclusive. Roll these, tracking statistics until you get a value twice in a row. Display `number of rolls`, `min`, `max`, and `average`.

## Claire is Where?

On New Year's Eve, have fun but don't forget your way home! For this challenge create four functions (`reset`, `moveBy`, `xLocation` and `yLocation`) to track the travels of Claire, a wanderer. Calling `reset()` moves Claire home to the origin `(0, 0)`. The `moveBy(xOffset, yOffset)` function moves her by those amounts, in those directions. Finally, `xLocation()` and `yLocation()` return how far Claire is from home, in X and Y directions respectively. After the calls of `reset()`, `moveBy(1, -2)`, and `moveBy(3, 1)`, subsequently calling `xLocation()` and `yLocation()` should return 4 and -1.

Second: create `distFromHome()`. Assuming she moves diagonally, return her distance from home.

## Chapter 2 – Fundamentals, Part II

---

### Date, on a Deserted Island

After a particularly fabulous New Year's Eve party to end 2016, Eduardo wakes to find himself stranded on a deserted island. He misses his home in Burbank, but at least now he can spend plenty of time outdoors – and you can't beat the commute! To pass the time until he is rescued, he counts sunrises.

- 1) Help Eduardo track what day of the week it is. Create a weekdayName (weekdayNum) function that, given a number between 1 and 7, will `console.log` a string containing the day of the week for that number (given 1, log "Sunday"). Use a `SWITCH` statement.
- 2) Expand weekdayName () to create weekdayName2 (dayNum) accepting numbers up to 365. Return weekday as before, given number of days total. "Sunday" still corresponds to 1.
- 3) Create a new function someDays () that calls weekDayName2 () seventeen times, with randomly generated integers as high as 365. Log each result string. If it is a weekday, add the phrase "**Work hard!**", and if it is a weekend day, add "**Enjoy your day off!**"
- 4) Build function monthName (monthNum) that, given a number from 1 to 12, returns a string containing month for that number ("May" corresponds to 5). Use an array, without loops.
- 5) Now expand monthName () to create monthToDays (monthNum), returning the number of days in that month, in the year 2017. Hint: use a `SWITCH` statement for the days in each month.
- 6) Despite using his *ember* expertise to create a glowing SOS visible from space, the days go by and sadly Eduardo is still not rescued. Is it spring yet? It might as well be. Build on monthName () to create dayToMonth (dayNum). If given a day number since the year began, return the current month (assume it is *not* a Leap Year). Given 75, return "March".
- 7) Eduardo builds a Dojo bootcamp on the island. Initially his students only find Ninja Gold in caves, but eventually even his tree sloths can write code quickly! Dojo classes meet Monday thru Friday, so let's reincorporate weekday to our calculations. Construct fullDate (dayNum) to accept number of days so far in 2017, and return a full date string. He hardly remembers that fateful New Year's Eve party, but he knows it was a Saturday. Given 142, return "Monday, May 22, 2017".
- 8) Times flies when you're at a Dojo – months in fact. Build fullDate2 (dayNum) that will be given a 4-digit integer: the days that have passed since December 31, 2016. This number can stretch into future years! You can assume that any year number divisible by four is a leap year and has a 29-day February. Given 8475, return "Thursday, March 15, 2040".
- 9) Eduardo hacks the Google Maps API and adds this long-forgotten island back onto the map. Soon he is rescued! News of his Hemingway-like stoicism make him famous for centuries. Build fullDate3 (dayNum) to handle days up to 140,000! Note: years 2100, 2200, and 2300 are not leap years (although 2400 *is*). Given 139947, return "Tuesday, February 29, 2400".

Having completed this epic saga, you are ready for additional concepts and chapters! Journey safely....

## Chapter 2 – Fundamentals, Part II

---

### Short Answer Questions: Fundamentals, Part II

What is the JavaScript `Math` library? What does it contain?

Why aren't all those functions just included in JavaScript automatically?

If I call `Math.random()`, what will it return?

What do the following functions do: `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round`?

When do `Math.floor` and `Math.trunc` *not* return the same value?

What is the `%` operator? When is it useful?

How do I make variables declared outside (right next to) a function *visible* inside it?

Why should I comment my code? If I know what it does, isn't commenting just a waste of time?

### Weekend Challenge: Fundamentals, Part II

This weekend, complete the “Date, on a Deserted Island” problem series, if you have not yet done so.

Then, if time allows, incorporate the `Math` library into a new quiz game. See if users can memorize 10 digits of Pi. Do the same for the constant e, the square root of 2, as well as the square root of 1/2.

These values are all available within the `Math` library. What other `Math` functions or properties can you incorporate? Continue to use `prompt`, and output the game statistics when the user quits.

### Fundamentals Part II Review

The two chapters prior to this one rapidly introduced you to many important concepts that underlie all of computing. These include *source code*, *programming languages*, *code flow*, *variables* and *data types* (such as numbers, booleans and strings), *conditionals*, *operators* (such as `=`, `==`, `====`, `&&`, `||`, `!`, `+=`, `++`, `-=`, `--` and `%`), *loops* (including `FOR`, `WHILE`, `BREAK`, `CONTINUE`), creating and executing your own *functions* (including input parameters and the return value), *arrays*, *T-diagrams* and *comments*.

This chapter added *numerical* tools to our palettes: including the `floor`, `ceil`, `round`, `trunc` and `random` functions from the `Math` library. We learned about the `%` operator, which is invaluable for determining remainders, or for extracting digits or fractional values from a compound number. We also were introduced to the concept of *scope*: a variable declared *within* a function is *not visible outside it*, but a variable declared *outside* a function *can be visible inside it*.

You are now fully equipped to solve a very broad number of basic computing challenges! There are still *many more* concepts for you to learn, so don't be discouraged if you go looking for interesting algorithm problems only to find that many of them are confusing or too difficult. Over time, with more repetition and practice, as these ideas sink in, and as we add more of these concepts, you will become better and better at breaking problems into understandable pieces that you can conquer with confidence. Onward!

There are various ways to test JavaScript code using artificial environments such as JSBin or plugins for your code editor. However, we prefer the real-world JS environment that *truly* matters: *browsers*. This way, you test your code in a way that doesn't just *emulate* the real-world: it *is* the real-world environment. Here are steps for one workflow that uses both editor and browser when creating/testing:

- 1) Go to Sublime/Atom, create a new file with the HTML file extension, and save it.
- 2) Within that file, type `html` and hit [tab]. Sublime will create the basic html tags for you.
- 3) Within the `<head>` section, type `script` and again hit [tab], to create `<script>` tags. You can delete the other attributes inside the `<script>` tag, as well as the `<style>` and even `<body>` tags; you don't *need* those if all you are doing is writing/testing JavaScript.
- 4) As the very first line after the `<script>` tag, always include `"use strict";`.
- 5) Now write your JS code as you wish. Remember to save as you go, of course.
- 6) When you are ready to test your code (which should be early and often), right-click on the sublime file and select "Open in Browser".
- 7) Your default browser will open a window. We prefer Chrome for testing JS, but any browser is fine. Within that window, open the Developer Console (the MacOS key sequence is *cmd-opt-J*).
- 8) Type JS into the console such as `returnGreaterThanSecond([1, 3, 5, 7, 9, 11, 15])` or `var arr=[]; returnGreaterThanSecond(arr)` or `printOdds()` to exercise your code.
- 9) To change or add to your code, cmd-tab back to Sublime/Atom, make your change, then Save.
- 10) Switch back to browser and Refresh, returning to step 8. You will soon memorize the quick-key sequence for saving in Sublime, returning to browser, and refreshing (cmd-s, cmd-tab, cmd-r).

Ultimately, your JavaScript code will run in the browser JS runtime environment, rather than an artificial environment such as JSBin. They produce slightly different results sometimes, so why bother....

This chapter explores the **array**: reading, changing, as well as adding and removing elements (which change the array's length). Before chapter's end, we touch on associative arrays as well. At this point we expect you to quickly complete the 13 mandatory algorithm challenges. Building array-inspection functions such as **min(arr)**, **max(arr)**, **sum(arr)** and **average(arr)** should be easy and rapid.

Let's review. Arrays store multiple values, which are accessed by specifying an *index* (offset from array front) in square brackets. This *random-access* makes arrays well-suited to be read in a different order than they were added. Arrays are less suitable (still common) in scenarios with many insertions and removals, if you need the array to stay in a particular sorted order. In that case, other values might need to be moved to create a space for inserting a new value (or, to fill a vacancy caused by removing a value). Arrays are not limited to one data type: one array can contain numbers, booleans, strings, etc.

Arrays are zero-based: *an array's first value is located at index 0*. Accordingly, array attribute **.length** literally means "one more than the last populated index." Like other interpreted languages, JavaScript arrays are not fixed-length; they automatically grow as values are set beyond the current length.

Tracking variables with T-diagrams can be very helpful when working with arrays. Use a T-diagram for this chapter's challenges. Below are constructs we'll use this chapter. Remember these building blocks!

## Declaring a new array:

```
var myArr = [];
console.log(myArr.length); // -> "0"
```

## Setting and accessing array values:

```
myArr[0] = 42;           // myArr == [42], length==1
console.log(myArr[0]);   // -> "42"
```

## Array.length is determined by largest index:

```
myArr[1] = "hi";        // myArr == [42,"hi"], length==2
myArr[2] = true;        // myArr == [42,"hi",true], length==3
```

## Arrays can be sparsely populated:

```
myArr[myArr.length+1] = 2; // myArr == [42,"hi",true,undefined,2]
console.log(myArr.length); // -> "5"
```

## Overwriting array values:

```
myArr[0] = 101;          // myArr == [101,"hi",true,undefined,2]
myArr[3] = "MG";         // myArr == [101,"hi",true,"MG",2]
```

## Shorten arrays with pop(), lengthen with push():

```
myArr.pop();             // myArr == [101,"hi",true,"MG"]
console.log(myArr.length); // -> "4"
myArr.push("dat");       // myArr == [101,"hi",true,"MG","dat"]
console.log(myArr.length); // -> "5"
```

# Chapter 3 – Arrays

---

From your work with the Basic 13 challenges, we assume that you already know how to read from numerical arrays, and that you can easily create JavaScript functions to get the **minimum** or **maximum** value, the **sum** of all values in the array, or the **average** of all values in the array. If this is not the case, definitely review those implementations before continuing to today's challenges.

Here is a list of concepts to consider; some or all will be used in this chapter.

<i>for / while loops</i>	<i>Array.pop()</i>	<i>Array.push()</i>
	<i>arrays grow: Array.length == lastIdx-1</i>	<i>if / else statements</i>
<i>arrays can contain different data types</i>		<i>arrays are objects, passed by reference (ptr)</i>

## Array: Push Front

Given array and an additional value, *insert this value at the beginning of the array*. Do this without using any built-in array methods.

## Array: Pop Front

Given array, *remove and return the value at the beginning of the array*. Do this without using any built-in array methods except **pop()**.

## Array: Insert At

Given array, index, and additional value, *insert the value into array at given index*. Do this without using built-in array methods. You can think of **pushFront(arr, val)** as equivalent to **insertAt(arr, 0, val)**.

## Array: Remove At

Given array and an index into array, *remove and return the array value at that index*. Do this without using built-in array methods except **pop()**. Think of **popFront(arr)** as equivalent to **removeAt(arr, 0)**.

## Array: Swap Pairs

Swap positions of successive pairs of values of given array. If length is odd, do not change the final element. For `[1, 2, 3, 4]`, return `[2, 1, 4, 3]`. For example, change input `["Brendan", true, 42]` to `[true, "Brendan", 42]`. As with all array challenges, do this without using any built-in array methods.

## Array: Remove Duplicates

Sara is looking to hire an awesome web developer and has received applications from various sources. Her assistant alphabetized them but noticed some duplicates. Given a sorted array, remove duplicate values. Because array elements are already in order, all duplicate values will be grouped together. As with all these array challenges, do this without using any built-in array methods.

**Second:** solve this without using any nested loops.

# Chapter 3 – Arrays

---

## Array.length

Some programmers think of `Array.length` as the number of array elements. This is *usually* true, but certain things can cause it *not* to be the case. If you create and control an array (as opposed to working with an array received from some other code), then you can avoid these things, which will make your code less complex. Let's explore `.length`, so that we can keep life simple, for you and your arrays.

The array property `.length` is defined as '*one greater than the largest populated index*':

```
var myArr == [42,"hi"];      // myArr.length == 2
myArr.push(true);           // myArr == [42,"hi",true], now length == 3
```

However, when directly setting values in arrays, we can add them at any (non-negative integer) index:

```
myArr[myArr.length+1] = 2; // myArr == [42,"hi",true,undefined,2]
console.log(myArr.length); // "5", although we set only 4 values
myArr.pop();              // myArr == [42,"hi",true,undefined]
console.log(myArr.length); // "4", although we never set myArr[3]
```

By setting an array value at an index beyond array's end, we created an empty space in our array – you could call it 'sparse' rather than having entirely contiguous values. Sparseness can be useful – in fact with associative arrays (objects), they will *always* be sparse – but generally with numerical arrays it vastly simplifies things to avoid this. How would you do this?

When adding a value to an array, use the `push()` function, or directly add the value to array's end (`arr[arr.length]`), or move another value there if you need the new value to be somewhere other than the array's end. In other words, if you don't use `push()`, make sure that `arr[arr.length]` is the next index in the array to be populated, rather than a larger index.

Likewise, when removing array values, use `pop()` or directly decrement the length (`arr.length--`). This means that if you need to remove a value from the middle of your array, you need to move the last value in the array into that middle index. Even though you are removing a middle value, you won't be removing that value's "chair". You'll actually be removing the last "chair" in the array, so the value currently there needs to be moved somewhere else!

Let's try these techniques on a few challenges. Remember, solve them without creating new arrays.

### Array: Min to Front

Given an array of comparable values, move the lowest element to array's front, shifting backward any elements previously ahead of it. Do not otherwise change the array's order. Given `[4,2,1,3,5]`, change it to `[1,4,2,3,5]` and return it. As always, do this without using built-in functions.

# Chapter 3 – Arrays

---

## Passing By Reference

Arrays are passed by *reference*. This means that when an array is sent as an argument, a pointer is sent. For this reason, even though parameters are always copies of the originals, with arrays (and all objects) a *pointer* is copied, resulting in caller and callee both having a copy of the same pointer. Hence both are looking at the same location in memory, and both will reference the same array. When we pass an array to another function, the array is passed “live” – changes the callee makes in that array are reflected when we return to the caller, regardless of whether the called function *returns* that array.

## Array: Reverse

Given a numerical array, reverse the order of values, in-place. The reversed array should have the same length, with existing elements moved to other indices so that order of elements is reversed. Working ‘in-place’ means that you *cannot use a second array* – move values within the array that you are given. As always, do not use built-in array functions such as `splice()`.

## Array: Rotate

Implement `rotateArr(arr, shiftBy)` that accepts array and offset. Shift `arr`’s values *to the right* by that amount. ‘Wrap-around’ any values that shift off array’s end to the other side, so that no data is lost. Operate in-place: given `([1, 2, 3], 1)`, change the array to `[3, 1, 2]`. Don’t use built-in functions. **Second:** allow negative `shiftBy` (shift L, not R).

**Third:** minimize memory usage. With no new array, handle arrays/`shiftBys` in the millions.

**Fourth:** minimize the touches of each element.

## Array: Filter Range

Alan is good at breaking secret codes. One tool is to eliminate numbers he knows are outside a certain specific range. Given `arr` and values `min` and `max`, remove array values between `min` and `max`. Work in-place: return the array you are given, with values in original order. No built-in array functions.

## Array: Concat

Replicate JavaScript’s `concat()`. Create a standalone function that accepts two arrays. Return a *new* array containing the first array’s elements, followed by the second array’s elements. Do not alter the original arrays. Ex.: `arrConcat( ['a', 'b'], [1, 2] )` should return new array `['a', 'b', 1, 2]`.

## Skyline Heights

Lovely Burbank has a breathtaking view of the Los Angeles skyline. Let’s say you are given an array with heights of consecutive buildings, starting closest to you and extending away. Array `[-1, 7, 3]` would represent three buildings: first is actually out of view below street level, behind it is second at 7 stories high, third is 3 stories high (hidden behind the 7-story). You are situated at street level. Return array containing heights of buildings you can see, in order. Given `[-1, 1, 1, 7, 3]` return `[1, 7]`. Given `[0, 4]` return `[4]`. As always with challenges, do not use built-in array functions such as `unshift()`.

# Chapter 3 – Arrays

## Another T-Diagram (Loops)

When working through a set of looping instructions with an array, a T-diagram can be especially useful. Let's do exactly that, with the following code and input array.

```
var arr = [42, 68, 7, 21, 243, 512];
for (var x = arr.length-2; x > 1; x--) {
    arr[x - 1] = arr[x + 1];
}
console.log(arr);
console.log(x);      // Will this work? Does x exist out here?
```

As we enter the loop for the first time, here is our T-diagram:

arr	[42, 68, 7, 21, 243, 512]
arr.length	6
x	4

We enter the loop (because  $x > 1$ ) then set `arr[x-1]` (or `arr[3]`, which is currently 21) to become `arr[x+1]`

(`arr[5]`, or 512). Accordingly, within our diagram in the line for `arr`, we change the 21 to 512. Then `x` decrements, so we change that value in our diagram as well, from 4 to 3. Because we see that we will reference `arr[x+1]` repeatedly, we add that to our T-diagram as well, with a current value of 243. Our updated T-diagram is as follows, as we return to loop's beginning, to evaluate whether to reenter:

Because  $x > 1$ , we indeed reenter the loop. We now set `arr[x-1]` (`arr[2]`, currently 7) to become `arr[x+1]`, which is already in our diagram (243). Then `x` decrements to 2, so we must update our `arr[x+1]` reference: it is `arr[3]`, which is 512. We now return to the `FOR` loop's beginning. To the right is our updated T-diagram, as we check whether to reenter the loop:

You can probably complete the exercise on your own at this point, but we'll continue onward for completeness. Again  $x > 1$ , so we do in fact reenter. Again we set `arr[x-1]` (`arr[1]`, currently 68) to become `arr[x+1]` (512 in our diagram). We decrement `x` to 1 and update our `arr[x+1]` reference to 243 (although ultimately we won't need this). Returning to our `FOR` loop's beginning, here is our updated T-diagram at this point:

Looking now at `x`, we see we will not reenter the `FOR`. We move on to `console.log` both `arr` and `x`. The final value of `arr` is [42, 512, 243, 512, 243, 512]. Yes, `x` has meaning outside the `FOR`, and its final value is 1.

We hope this second walkthrough shows the clarity that T-diagrams can bring when iterating arrays.

# Chapter 3 – Arrays

---

Here are the concepts/methods we've discussed; some or all will be used in this chapter's challenges. As always, don't use built-in array methods.

*for / while loops*

*Array.pop() & push()*

*avoid sparseness*

*arrays grow: arr.length == lastIdx-1*

*if / else statements*

*can contain different data types*

*arrays are objects, passed by reference (ptr)*

## Array: Remove Negatives

Implement `removeNegatives()` that accepts an array, removes negative values, and returns the same array (not a copy), preserving non-negatives' order. As always, do not use built-in array functions.

**Second:** don't use nested loops.

## Array: Second-to-Last

Return the second-to-last element of an array.

Given `[42, true, 4, "Liam", 7]`, return  
"Liam". If array is too short, return `null`.

## Array: Second-Largest

Return the second-largest element of an array.

Given `[42, 1, 4, Math.PI, 7]`, return 7. If the array is too short, return `null`.

## Array: Nth-to-Last

Return the element that is N-from-array's-end.

Given `([5, 2, 3, 6, 4, 9, 7], 3)`, return 4. If the array is too short, return `null`.

## Array: Nth-Largest

Jimmy has "N" number of Green Belt stickers for excellent Python projects. Given `arr` and `N`, return the `N`th-largest element, where `(N-1)` elements are larger. Return `null` if needed.

## Credit Card Validation

The Luhn formula is sometimes used to validate credit card numbers. Create the function `isCreditCardValid(digitArr)` that accepts an array of digits on the card (13-19 depending on the card), and returns a boolean whether the card digits satisfy the Luhn formula, as follows:

- 1) Set aside the last digit and do not include it in these calculations (until step 5);
- 2) Starting from the back, multiply the digits in odd positions (last, third-to-last, etc.) by 2;
- 3) If any results are larger than 9, subtract 9 from them;
- 4) Add all numbers (not just our odds) together;
- 5) Now add the last digit back in – the sum should be a multiple of 10.

For example, when given digit array `[5, 2, 2, 8, 2]`, after step 1) it becomes `[5, 2, 2, 8]`, then after step 2) it is `[5, 4, 2, 16]`. Post-3) we have `[5, 4, 2, 7]`, then following 4) it becomes 18. After step 5) our value is 20, so ultimately we return `true`. If the final digit were any non-multiple-of-10, we would instead return `false`.

# Chapter 3 – Arrays

---

## **“Truthy” and “Falsey”**

JavaScript is well known for its ‘loose’ treatment of data types. In actuality, JavaScript considers almost everything *an object*, since almost every possible value has a set of methods attached (`valueOf`, `toString`, etc). That said, `typeof` returns six possible values, suggesting that there are six top-level data types: `boolean`, `number`, `string`, `object`, `function`, and `undefined`. (A seventh, introduced in ES6, is left as an exercise for the reader.) JavaScript converts values between data types, as needed. For example, `if()` converts any value to a boolean, to decide which way to branch.

Most values are considered *something* and if converted to a boolean, equate to `true`. Only six values are considered *nothing*: `false`, `0`, `NaN`, `""`, `null`, `undefined`; these six are “falsey”, because when converted to a boolean, they equate to `false`. All other values are “truthy”, including all functions, objects, non-0 numbers (e.g.: `-Infinity`) and non-empty strings (e.g.: `"0"` or `"false"`).

### **□ Array: Shuffle**

In JavaScript, the `Array` object has numerous useful methods. It does not, however, contain a method that will randomize the order of an array’s elements. Let’s create `shuffle(arr)`, to efficiently shuffle a given array’s values. Work in-place, naturally. Do you need to return anything from your function?

### **□ Array: Remove Range**

Given array, and indices `start` and `end`, remove vals in that index range, working in-place (hence shortening the array). Given `([20,30,40,50,60,70], 2, 4)`, change to `[20,30,70]` and return it.

### **□ Intermediate Sums**

You will be given an array of numbers. After every tenth element, add an additional element containing the sum of those ten values. If the array does not end aligned evenly with ten elements, add one last sum that includes those last elements not yet been included in one of the earlier sums. Given the array `[1,2,1,2,1,2,1,2,1,2,1,2,1,2]`, change it to `[1,2,1,2,1,2,1,2,1,2,15,1,2,1,2,6]`.

### **□ Double Trouble**

Create a function that changes a given array to list each existing element twice, retaining original order. Convert `[4, "Ulysses", 42, false]` to `[4, 4, "Ulysses", "Ulysses", 42, 42, false, false]`.

### **□ Zip It**

Create a standalone function that accepts two arrays and combines their values sequentially into a new array, at *alternating indices* starting with first array. Extra values from either array should be included afterward. Given `[1,2]` and `[10,20,30,40]`, return new array containing `[1,10,2,20,30,40]`.

**Second:** combine the two arrays’ values *into the first array*, instead of into a new array. Much more fun!

# Chapter 3 – Arrays

---

## Short Answer Questions: Arrays

What is an array?

What types of values can be held in an array?

What happens if you try to contain values of different data types in an array?

What is the index of the first element in an array?

What is the index of the last element in an array?

If you control an array, what is the easiest way to determine the number of values?

When would the above method *not* work well, and what must you do in that case?

What are two ways to make an array one element longer?

What are two ways to make an array two elements shorter?

What is the upper limit on the number of values an array can hold?

When working with arrays, does working “in-place” really matter? If so, when and why?

Which values are “falsey” in JavaScript, and what does “falsey” mean anyway?

What does “passing by reference” mean?

## Weekend Challenge: Arrays

For this weekend challenge, consider the game Tic-Tac-Toe. How would you represent a 3x3 game board: specifically whether each square is unoccupied or claimed by a player? Could you represent this with an array? What would the array contain? Is there a benefit from having an array of arrays?

Once you have a good design in mind, create a function that, given one of these gameboards, automatically determines and returns the overall state of the game. A game is always in one of five different states – one state might be “Player 1’s turn”; another might be “Game over: player 2 has won.”

Once you have mastered that, can you extend this function to handle gameboards of arbitrary size?

## Arrays Review

This chapter covered JavaScript arrays in more depth. We showed how to declare and initialize arrays, how to read from specific indices, and how to write values into arrays – both overwriting existing values, as well as writing into new index locations that likely extend an array’s length. We made significant use of the property .length that is present in every array, and we discussed how this property is usually (but *not always*) equal to the number of elements present in the array. Specifically, we mentioned that arrays can be sparse, which means we can configure an array so that certain index locations have not yet been written with any value (and hence contain undefined). We rehearsed numerous times the iteration of an array, using a FOR loop. We touched on the subject of function parameters that are passed by reference, and how that changes a function from purely returning advisory information to making permanent changes in the array (or other parameter). We did an exhaustive walkthrough of debugging FOR loops with arrays. Finally, we built on our new understanding of JavaScript data types, and discussed the values that (across type conversion) all equate to false – the six “falsey” values.

# “Basic 13” Review

Solutions for the “Basic 13” algorithm challenges.

## Print 1-255

Print all the integers from 1 to 255.

```
function print1to255()
{
    var num = 1;
    while (num <= 255) {
        console.log(num);
        num = num + 1;
    }
}
```

## Print Ints and Sum 0-255

Print integers from 0 to 255, and the sum so far.

```
function printIntsAndSum0to255()
{
    var sum = 0;
    for (var num = 0; num <= 255; num++) {
        sum += num;
        console.log("New number:" + num + "Sum:" + sum);
    }
}
```

## Find and Print Max

Print the largest element in a given array.

```
function printArrayMax(arr)
{
    if (arr.length == 0) {
        console.log("Empty array, no max value.");
        return;
    }

    var max = arr[0];
    for (var idx = 1; idx < arr.length; idx++) {
        if (arr[idx] > max) {
            max = arr[idx];
        }
    }
    console.log("Max value is:" + max);
}
```

# “Basic 13” Review

---

Solutions for the “Basic 13” algorithm challenges.

## Print Odds 1-255

Print all odd integers from 1 to 255.

```
function printOdds1to255() {  
    var num = 1;  
    while (num <= 255) {  
        console.log(num);  
        num = num + 2;  
    }  
}
```

## Array with Odds

Create an array with odd integers from 1-255.

```
function oddArray1to255() {  
    var oddArray = [];  
    for (var num = 1; num <= 255; num += 2)  
    {  
        oddArray.push(num);  
    }  
    return oddArray;  
}
```

## Iterate and Print Array

Print all values in a given array.

```
function printArrayValues(arr) {  
    for (var index = 0; index < arr.length; index++) {  
        console.log("array[" + index + "] is equal to" + arr[index]);  
    }  
}
```

## Get and Print Average

Analyze an array's values and print the average.

```
function printArrayAverage(arr) {  
    if (arr.length == 0) {  
        console.log("Empty arr, no average val");  
        return;  
    }  
  
    var sum = arr[0];  
    for (var idx = 1; idx < arr.length; idx++) {  
        sum += arr[idx];  
    }  
    console.log("Average value is:" + sum / arr.length);  
}
```

# “Basic 13” Review

---

Solutions for the “Basic 13” algorithm challenges.

## Greater than Y

Count and print the number of array values less than a given Y.

```
function numGreaterThanY(arr, y) {  
    var numGreater = 0;  
    for (var idx = 0; idx < arr.length; idx++) {  
        if (arr[idx] > y) { numGreater++; }  
    }  
    console.log("%d values are greater than %d", numGreater, y);  
}
```

## Max, Min, Average

Given an array, print max, min and average values.

```
function maxMinAverage(arr) {  
    if (arr.length == 0) {  
        console.log("Null arr, no min/max/avg");  
        return;  
    }  
    var min = arr[0];  
    var max = arr[0];  
    var sum = arr[0];  
    for (var idx = 1; idx < arr.length; idx++) {  
        if (arr[idx] < min) { min = arr[idx]; }  
        if (arr[idx] > max) { max = arr[idx]; }  
        sum += arr[idx];  
    }  
    console.log("Max:" + max + " Min:" + min);  
    console.log("Avg value:" + sum / arr.length);  
}
```

## Square the Values

Given an array, square each value in the array.

```
function squareArrVals(arr) {  
    for (var idx = 0; idx < arr.length; idx++) {  
        arr[idx] = arr[idx] * arr[idx];  
    }  
    return arr;  
}
```