khalifenizar / **technical-interview-prework.md** SECRET

Last active a month ago

Ironhack Technical Interview Pre-work

Ironhack

Table of Contents

- Introduction
- JavaScript
- Hello, world!
- Values
- Comments
- Math and other operations
- Saving values
- Making your programs smart
- Repetitive tasks
- Packaging code
- Structuring values
- Additional exercises

Technical Interview Pre-work

Hello, prospective student! If you are reading this document that means that you've got a bit of work to do before you have your technical interview. Don't worry! We at Ironhack are here to help, not hinder you. Hopefully after going through this content you will feel more confident in your skills, enough to destroy our technical interview problem and get accepted into Ironhack!

Let's get started!

JavaScript

This content will help you familiarize yourself with basic programming concepts in JavaScript. JavaScript is a programming language that runs in Web browsers. To write and run your code you can use repl.it, the open source online programming environment.

Hello, world!

Traditionally the first program one writes is always a greeting to the world of programming: *Hello, world!* In JavaScript it looks

a little something like this:

```
console.log("Hello, world!");
```

Paste that into `repl.it` and run it to see what you get.

What did I just do?

Basically the way you print out messages in a JavaScript program is with a *function* called `console.log`. We will talk about functions a bit later, but suffice to say that you can use `console.log` to print out any message you want. All you need to do is type `console.log(` (with parentheses), your message *within quotes* and finally a closing parentheses and semi-colon: `);`. Try different messages now:

```
console.log(  "This needs to be in quotes or it won't work!"  );
console.log(  "Help! I'm trapped in the computer."  );
```

Exercise

Now try not using quotes or leaving out one of the parentheses. What happens?

Values

The reason the quotes are important when printing your message is because JavaScript can handle different kinds of **values**. When you use quotes you are telling JavaScript that you want to use a *text* value. Text values are known as **strings** in programming. An empty string value is valid (although not very useful to print out). Try it:

```
console.log("");
```

Another very common value is a *number*. You don't need to use quotes for numbers. Try it on `repl.it`:

```
console.log(9999);
```

Comments

Comments are parts of the code that you, the programmer, tell JavaScript to ignore. You can use two slashes `//` to have JavaScript ignore the rest of that line. Try putting a comment before and after printing a message:

```
// This is a commment
// This is a second comment
console.log("This is a message");  // This is a third comment
```

What's the point of that? Well comments are used by program authors to leave messages for other programmers. We will leave explanatory messages in some of our code examples for your benefit!

Math and other operations

Values alone aren't so useful until you can start to doing work with them. JavaScript has a few built-in ways to work with values. These are called **operators**. Basic math such as addition and subtraction are good examples of **operators**. Try some math and print out the results:

```
// Addition +
console.log(  10 + 5  );

// Subtraction -
console.log(  7 - 3  );

// Multiplication *
console.log(  6 * 7  );

// Division /
console.log(  8 / 2  );
console.log(  5 / 2  );

// Remainder %
console.log(  6 % 3  );
console.log(  8 % 3  );
```

Strings also have a basic text operation: combining text using the plus sign `+`. Try it:

```
console.log(  "Iron" + "hack"  );
```

In programming, combining strings is called **concatenation**.

Exercise

Now we've seen that the plus sign is used for both number addition and string concatenation. What happens when you add a number to a string?

Saving values

Often you are going to want to save a value. There are two main reasons to do that:

1. You want to re-use the value in several places in your code. Having to re-type long values is tedious and it makes it easier to make a typo.
2. You want to save the value and change it slowly over the execution of your program.

You save a value in a `variable`. In JavaScript you first need to *declare* a variable once and then you can use it as many times as you want. You declare a variable by typing `var` and the name of the variable. To save a value in the variable you use the equal sign `=`. Try it:

```
var city;

city = "Madrid";

console.log(city);

city = "Barcelona";

console.log(city);


var country = "Spain";

console.log(country);

country = "United States";

console.log(country);
```

```
country = country + " of America";

console.log(country);
```

Exercise

Try experimenting with different variable names.

1. Can you use numbers in your variable name (`country13`)?
2. Can a variable name start with a number?
3. Can you use characters like `_` , `&` , `^` , `~` , `-` , `$` and `#` in variable names? Try them all.

Making your programs smart

Basic comparisons

To really make your programs interesting they need to start thinking for you! A simple way to do that is checking if values meet certain conditions. To do that you use `if` structures in combination with special **comparison operators**. If the condition you specify with comparison operators is met, the instructions inside the `if` are run. Try it:

```
var number = 10;

// Equal
if (number === 10) {
    console.log("Number is equal to 10!");
    console.log(number);
}

number = number - 1;

// Not equal
if (number !== 10) {
    console.log("Number is now NOT equal to 10!");
    console.log(number);
}

// Less than
if (number < 10) {
    console.log("Number is less than 10!");
    console.log(number);
}

number = number + 2;

// Greater than
if (number > 10) {
    console.log("Number is now greater than 10!");
    console.log(number);
}

// Greater than or equal
if (number >= 10) {
    console.log("Number is greater than or equal to 10!");
    console.log(number);
}

number = 10;

if (number >= 10) {
```

```
    console.log("Number is STILL greater than or equal to 10!");
    console.log(number);
}

// Less than or equal
if (number <= 10) {
    console.log("Number is less than or equal to 10!");
    console.log(number);
}

number = number - 2;

if (number <= 10) {
    console.log("Number is STILL less than or equal to 10!");
    console.log(number);
}
```

Comparing strings

Strings use **alphabetical order** for comparison, capital letters being ordered first:

```
var country;

country = "Australia";

if (country < "Belgium") {
    console.log(country + " is less than Belgium!");
}

country = "australia";

if (country > "Australia") {
    console.log(country + " is greater than Australia!");
}
```

Combining camparisons

You can also **combine** different comparisons to make a larger, more specific one using `&&` and `||`:

```
var country, number;

country = "Australia";

// || represents an OR
if (country === "Australia" || country === "New Zealand") {
    console.log(country + " is either Australia or New Zealand!");
}

country = "New Zealand";

if (country === "Australia" || country === "New Zealand") {
    console.log(country + " is either Australia or New Zealand!");
}

number = 15;

// && represents an AND
if (number > 10 && number < 20) {
    console.log(number + " is between 10 and 20.");
}
```

```
number = 21;

// No longer runs because number is greater than 20
if (number > 10 && number < 20) {
    console.log(number + " is between 10 and 20.");
}
```

Doing things when `if` doesn't match

To catch cases that don't match your `if` comparison you open an `else`. Try it:

```
var number;

number = 21;

if (number > 10 && number < 20) {
    console.log(number + " is between 10 and 20.");
} else {
    console.log(number + " is NOT between 10 and 20.");
}
```

You can also combine `else` with another `if` to make more specific logic:

```
var number;

number = 21;

if (number > 10 && number < 20) {
    console.log(number + " is between 10 and 20.");
} else if (number <= 10) {
    console.log(number + " is less than 10.");
} else {
    console.log(number + " is greater than 20.");
}
```

Keep in mind that when using chains of `if` and `else` only the section that matches your case will run. The other sections of code will be ignored.

Exercise

1. How do characters like `!`, `@`, `#`, `$`, `%`, `^`, `&` and `*` affect string comparison? Which is greater: `Argentina` or `!Argentina`?
2. This comparison probably doesn't work like you expect it to. Can you figure out why not?

```
var number;

number = 101;

if (number !== 101 && number < 25 || number > 100) {
    console.log(number + " isn't 101 AND is less than 25 OR is greater than 100.");
}
```

Repetitive tasks

Another thing that you will certainly want to do in your programs is perform repetitive tasks. The simplest way to accomplish this is with **loops**. Loops are blocks of code that run over and over again until a certain condition is met. The condition can be

anything you can think of! Just make sure that part of your loop includes code that changes the condition so that it is eventually met. Let's take a look at a loop using `for`:

```
var i;

for (i = 1;  i <= 42;  i = i + 1) {
    console.log(i);
}
```

Huh?

So what's going on there? Here's the structure of a `for`:

```
for (  [start];  [condition];  [after-each]  ) {
    [task-code]
}
```

Basically a `for` loop wants to run `[task-code]` a bunch of times. In our example the only code we want to run a bunch of times is printing the variable `i`. As for that other weird stuff:

- `[start]` is code to run to set some initial state before your loop starts. In our example we just set the initial value of `i` to `1`.
- `[condition]` is the comparison to perform to decide whether or not to keep running the loop again. When `i` becomes greater than `42` our loop will stop.
- `[after-each]` is the code to run after each time we go through the loop to change the condition. This is supposed to ensure that eventually the condition will not be met and so the loop will stop at some point. In our case, we are adding `1` to `i` so that we will get closer to `42` every time we go through the loop. *Be careful* because JavaScript *will* let you write a loop that never stops (we call them **infinite loops**).

Exercise

Write a `for` loop that sings! Starting with one, each run through the loop should print a bunch of `la` until you print out `lalalalalalalalala`. Like this:

```
la
lala
lalala
...
lalalalalalalalala
```

Packaging code

Sometimes you will find the need to package some pieces of your code into self-contained chunks so that you can use them easily in different parts of your program. You can do this by writing a `function`. Let's look at an example first:

```
function claimForCountry (country, thing) {
    var claim;

    claim = "In the name of " + country + " I claim this " + thing + "!";

    return claim;
}

var message;
```

```
message = claimForCountry("Spain", "land");

console.log(message);

message = claimForCountry("Italy", "pizza");

console.log(message);
```

`function` structure

```
function [name] ( [parameter1], [parameter2], [parameter3]... ) {
    [code]
    return [value];
}
```

Let's break down `function`:

- A `function` has a `[name]` that you call it by. In our example that name is `claimForCountry`.
- When you call it you send it as many `[parameters]` as it needs. Parameters are just values you send the function. Our example `claimForCountry` needs two parameters:
  - `country`: The name of the country that is making the claim.
  - `thing`: The name of the thing that is being claimed.
- Then the `function` runs `[code]` with those parameters. In our example we declare a variable and set it's value using `country` and `thing`.
- Finally, after running the code it **returns** a `[value]`. In our case we return the `claim` variable's value.

So when you call `claimForCountry` it returns a string value which you can use as you see fit. You can also send it different parameters to change the message.

Variations

The structure of function is not super strict. There are certain parts that are optional. For example, you can write a function with as many or as few parameters as you want. Try it:

```
// No parameters
function hi () {
    return "Hi!";
}

console.log( hi() );

// Five parameters
function addFiveNumbers (n1, n2, n3, n4, n5) {
    return n1 + n2 + n3 + n4 + n5;
}

console.log( addFiveNumbers(10, 20, 30, 5, 7) );
```

Returning a value is also optional. For example we could have written the `hi` function to use `console.log` directly:

```
// No return
function hi () {
    console.log("Hi!");
}

hi();
```

However, keep in mind that you may want to do different things with values besides `console.log` so it's often best to `return`.

Exercise

1. Remember we said `console.log` is a function. Can it handle more than one parameter? How many parameters can it accept?
2. Write a function that receives a number and prints a countdown of that number all the way to 0 and then `Blast off!`. For example, `blastOff(3)` would print this:

```
3
2
1
0
Blast off!
```

Structuring values

In your programming travels you will also have need to structure values into more complex patterns. Let's go over the two basic ways of doing that.

Arrays

An array is a list of values within square brackets `[]`. Let's see a quick example:

```
var countries;

countries = [ "Russia", "Finland", "Morocco", "Brazil" ];

console.log(countries);
```

Once you create your array, you can pass it around and use it like any other value. You could also think of an array as a row of lockers because each value has a number position assigned to it, starting with 0. You can access each position's value by using square brackets `[]`:

```
var countries;

countries = [ "Russia", "Finland", "Morocco", "Brazil" ];

// Russia
console.log("The first country is " + countries[0] + "!");

// Finland
console.log("The second country is " + countries[1] + "!");
```

This makes it convenient to use a `for` to loop through an array by using the `length` value attached to the array:

```
var countries, i;

countries = [ "Russia", "Finland", "Morocco", "Brazil" ];

for (i = 0; i < countries.length; i = i + 1) {
    console.log(countries[i]);
```

```
}
```

You can also use the `push` function attached to an existing array to add more values to it:

```
var countries;

countries = [];

countries.push( "Russia", "Finland", "Morocco", "Brazil" );
countries.push( "Canada" );
countries.push( "Mexico", "Japan" );

console.log(countries);
```

Objects

Objects are similar to arrays except they use curly braces `{}` for creating them and instead of using numbers to label values, you use strings. These strings are called **keys**. Objects are great for representing things from the real world in our code. Try it:

```
var country;

country = {
    name: "Uruguay",
    continent: "South America",
    capital: "Montevideo"
};

console.log(country);
```

You can access, add or change values in an object in two ways. The first is similar to arrays but using a string and the second is with a dot `.` between the object name and the key:

```
var country;

country = {
    name: "Uruguay",
    continent: "South America",
    capital: "Montevideo"
};

console.log( country["name"] );
console.log( country.continent );

country["language"] = "Spanish";
country.population = 3324460;
country.name = "Eastern Republic of Uruguay";

console.log(country);
```

Finally, you can loop over the keys in an object as well with a special `for` syntax called `for ... in`. Try it:

```
var country;

country = {
    name: "Uruguay",
    continent: "South America",
    capital: "Montevideo"
```

```
};

for (key in country) {
    console.log("This country's " + key + " is " + country[key] + ".");
}
```

When you are using `for ... in`, remember to use the square brackets `[]` for accessing values in an object. Using the dot won't do what you expect because `country.key` will look for a value in the object that's literally called `key`.

Exercise

1. What happens when you try to access a value that doesn't exist in arrays and an objects? Try it!
2. Write a function that receives an array of names and returns them in a string with `,` between them. The final two must be separated by `and`.
3. Write a function that receives an object and returns an array of its values.

Example for #2:

```
var names, sentenceNames;

names = [ "Canada", "Mexico", "Turkey", "Japan" ];

// Should be "Canada, Mexico, Turkey and Japan"
sentenceNames = prepare(names);
console.log(sentenceNames);
```

Example for #3:

```
var country, countryArray;

country = {
    name: "Uruguay",
    continent: "South America",
    capital: "Montevideo"
};

// Should be [ "Uruguay", "South America", "Montevideo" ]
countryArray = objectToArray(country);
console.log(countryArray);
```

Additional exercises

A. Sequence addition

Write a function that receives a number and loops from `1` to the number and on each run through adds the current and previous numbers and prints out the result. On the first run through (current number = `1`) use the value of `0` for the previous number. For example on this call:

```
addSequence(15);
```

You will loop from `1` to `15`:

- On `1`: print the result of `0 + 1`
- On `2`: print the result of `1 + 2`

- On `3` : print the result of `2 + 3`
- On `4` : print the result of `3 + 4`
- On `5` : print the result of `4 + 5`
- On `6` : print the result of `5 + 6`
- On `7` : print the result of `6 + 7`
- On `8` : print the result of `7 + 8`
- On `9` : print the result of `8 + 9`
- On `10` : print the result of `9 + 10`
- On `11` : print the result of `10 + 11`
- On `12` : print the result of `11 + 12`
- On `13` : print the result of `12 + 13`
- On `14` : print the result of `13 + 14`
- On `15` : print the result of `14 + 15`

B. Print strings with a minimum length

To get the length of a string you can use the `length` property:

```
var str = "Zanzibar";

console.log(str.length);
// => 8
```

Write a function that receives a number that is the minimum length and an array of strings. Loop through the array of strings and print only those that are equal to or above the minimum length. Once you have your `printMinimum()` function this code should work for you:

```
var strs = [
    "I like pie.", "Hello.", "Shorts.", "This is a long one.",
    "Kinda medium.", "A.", "Shorty."
];

printMinimum(10, strs);
// => I like pie.
//    This is a long one.
//    Kinda medium.

printMinimum(7, strs);
// => I like pie.
//    Shorts.
//    This is a long one.
//    Kinda medium.
//    Shorty.

printMinimum(2, strs);
// => I like pie.
//    Hello.
//    Shorts.
//    This is a long one.
//    Kinda medium.
//    A.
//    Shorty.
```

C. Secret message decoder

To retrieve a letter from a string you can use the `charAt()` function like so:

```
var str = "Zanzibar";

// Character at index 3 (the 4th character)
var fourth = str.charAt(3);
console.log(fourth);
// => "z"
```

As you can see, character positions in strings start at 0, just like in arrays. If we ask for the character at position `3`, we are really referring to the 4th character, `z`.

Write a function that decodes a secret message hidden in an array of words. Each word contains one letter of the message. Go through the words in order and retrieve one character from each of them. For the 1st word, grab the 1st character, for the 2nd word, the 2nd character and so on. When you get to the 6th word, start from the 1st character again. Take each of those characters and add them to a new string to form the message. Finally, return the message. Once you have your `decode()` function this code should work for you:

```
var words, message;

words = [
    "dead",         // 1st -> d
    "bygone",       // 2nd -> y
    "landing",      // 3rd -> n
    "cheaply",      // 4th -> a
    "assumed",      // 5th -> m
    "incorrectly",  // 1st -> i
    "attention",    // 2nd -> t
    "agent"         // 3rd -> e
];

// message should be "dynamite"
message = decode(words);
console.log(message);
```

Try to decode the messages hidden in these two arrays:

```
var words1 = [
    "January", "lacks", "caveats", "hazardous", "DOORS",
    "crying", "arrogantly", "climate", "proponent", "rebuttal"
];

var words2 = [
    "Issuing", "anecdotal", "reticles", "selecting", "recurring",
    "nodes", "belonging", "externally", "braziers"
];
```

D. Array looper

Write a function that receives an array and a string. Print all the values inside the array by looping through it. Depending on the value of the string you will loop through the array in one of four different ways. For example take this array:

```
var foods = [ "pizza", "cookies", "ice cream", "steak", "burgers", "bread" ];
```

Loop through the array normally if the string is equal to `forwards`:

```
looper(foods, "forwards");
// => pizza
//    cookies
```

```
//    ice cream
//    steak
//    burgers
//    bread
```

Loop through the array backwards if the string is equal to `backwards`:

```
looper(foods, "backwards");
// => bread
//    burgers
//    steak
//    ice cream
//    cookies
//    pizza
```

Ignore values with odd indexes if the string is equal to `evens` (the number `0` is considered even):

```
looper(foods, "evens");
// => pizza
//    ice cream
//    burgers
```

Ignore values with even indexes if the string is equal to `odds`:

```
looper(foods, "odds");
// => cookies
//    steak
//    bread
```

E. Add foods to an array

Write a function that receives an array of foods and checks if it contains the values "pizza" and "bacon". If either of those values is missing, add it to the array.

```
var foods1 = [ "cookies", "steak" ];
addFoods(foods1);
// foods1 should now be [ "cookies", "steak", "pizza", "bacon" ]

var foods2 = [ "pizza", "salad" ];
addFoods(foods2);
// foods2 should now be [ "pizza", "salad", "bacon" ]

var foods3 = [ "pizza", "salad" , "bacon", "chili" ];
addFoods(foods3);
// foods3 should still be [ "pizza", "salad" , "bacon", "chili" ]
```

Matthew23Smith commented on Nov 10

cool!! looks easy but it's clever XD

LadarrisSain commented on Nov 17

I'm having difficulty with Exercise C. I'm assuming that we are we attempting to create a function that "loops" through the array?

I've come up with this code for the first "words" array. Unfortunately, it does not contain a loop and only works when the initial "words" array is passed in. Eventually I was able to come up with a loop that began to work, however it

```
var words, message;

words = [
    "dead",         // 1st -> d
    "bygone",       // 2nd -> y
    "landing",      // 3rd -> n
    "cheaply",      // 4th -> a
    "assumed",      // 5th -> m
    "incorrectly",  // 1st -> i
    "attention",    // 2nd -> t
    "agent"         // 3rd -> e
];

function decode(words) {
  var first, second, third, fourth, fifth, sixth, seventh, eigth;
  var first = words[0].charAt(0);
  var second = words[1].charAt(1);
  var third = words[2].charAt(2);
  var fourth = words[3].charAt(3);
  var fifth = words[4].charAt(4);
  var sixth = words[5].charAt(0);
  var seventh = words[6].charAt(1);
  var eigth = words[7].charAt(2);
  console.log("The secret code is" + " " + first + second + third + fourth + fifth + sixth + seventh +
              eigth);
  return first + second + third + fourth + fifth + sixth + seventh +
              eigth;
};

// message should be "dynamite"
message = decode(words);
console.log(message);
```

This is the only loop I have been able to configure that begins working somewhat. I am having difficulty with making it restart at the index of 0 after it reaches 5 when looping through the strings in the array and pulling the appropriate letter to apply to the decoded message.

```
function decode(a) {
  for(i=0;i<=a.length;i++) {
   var z = a[i].charAt(i);
   console.log(z);
  }
};
```

This function prints out the message below when called.

```
d
y
n
a
m
r
i
TypeError: Cannot read property 'charAt' of undefined (line 22 in function decode) …
```

I've been working on this for a few days now and I am about stumped. Any hints?