Understanding Synchronous and Asynchronous JavaScript – Part 1

Published by Preethi Ranjit, in Coding

Synchronous and **asynchronous** are confusing concepts in JavaScript, especially for beginners. Two or more things are **synchronous** when they **happen at the same time** (in sync), and **asynchronous when they don't** (not in sync).

15 JavaScript Methods For DOM Manipulation for Web Developers

As a web developer, you frequently need to manipulate the DOM, the object model that is used by…

Read more

You might also like

20+ Sports-Related Freebies to Design for the Olympics
Hongkiat Lim

Freebie: 10 High Quality Summer Letterings
Hongkiat Lim

Infographic Elements By Vector Open Stock
Hongkiat Lim

Although these definitions are easy to take in, it is actually more complicated than it looks. We need to take into account **what exactly are in sync**, and **what aren't**.

You'd probably call a "normal" function in JavaScript synchronous, right? And if it's something like `setTimeout()` or AJAX that you're working with, you will refer to it as being asynchronous, yes? What if I tell you that ***both* are asynchronous in a way**?

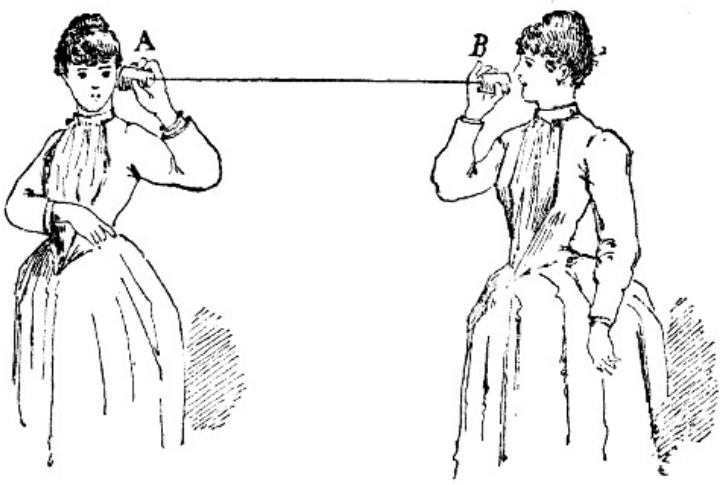
To explain the *why*, we need to turn to Mr X for help.

Scenario 1 – Mr X is trying synchronicity

Here's the setup:

1. Mr X is someone who can answer tough questions, and carry out any requested task.
2. The only way to contact him is through a phone call.
3. Whatever question or task you got, in order to ask Mr X's help to carry it out; you call him.
4. Mr X gives you the answer or completes the task **right away**, and lets you know **it's done**.
5. You put down the receiver feeling content and go out for a movie.

What you've just carried out was a **synchronous (back and forth) communication** with Mr X. He listened as you were asking him your question, and you listened when he was answering it.



Scenario 2 – Mr X isn't happy with synchronicity

Since Mr X is so efficient, he starts receiving many more calls. So what happens when you call him but **he's already busy** talking to someone else? You won't be able to ask him your question – not till he is free to receive your call. All you will hear is a busy tone.

So what can Mr X do to combat this?

Instead of taking calls directly:

1. Mr X hires a new guy, Mr M and gives him an answering machine for the callers **to leave messages**.
2. Mr M's job is to **pass on a message** from the answering machine to Mr X once he knows Mr X has completely finished processing all previous messages and is already **free to take a new one**.
3. So now when you call him, instead of getting a busy tone, you get to leave a message for Mr X, then **wait for him to call you back** (no movie time yet).
4. Once Mr X is done with all the queued up messages he received before yours, he will look into your issue, and **call you back** to give you an answer.

Now here lies the question: were the actions so far **synchronous or asynchronous?**

It's mixed. When you left your message, **Mr X wasn't listening in to it, so the forth communicatio onous.**

But, when he replied, **you were there listening**, which **makes the return communication synchronous**.

I hope by now you have acquired a better understanding of how synchronicity is perceived in terms of communication. Time to bring in JavaScript.

JavaScript – An Asynchronous Programming Language

When someone labels JavaScript asynchronous, what they are referring to in general is how you can **leave a message** for it, and **not have your call blocked** with a busy tone.

The function calls are **never direct in JavaScript**, they're literally done **via messages**.

JavaScript uses a **message queue** where incoming messages (or events) are held. An **event-loop** (a message dispatcher) sequentially dispatches those messages to a **call stack** where the corresponding functions of the messages are **stacked as frames** (function arguements & variables) for execution.

The call stack holds the frame of the initial function being called, and any other frames for functions called **via nested calls** on top of it .

When a message joins the queue, it waits until the call stack is **empty of all frames from the previous message**, and when it is, the event-loop **dequeues the previous message**, and adds the corresponding frames of the current message to the call stack.

The message waits again until the call stack becomes **empty of its own corresponding frames** (i.e. the executions of all the stacked functions are over),

then is dequeued.

Consider the following code:

```
function foo(){}
function bar(){
  foo();
}
function baz(){
  bar();
}
baz();
```

The function being run is `baz()` (at the last row of the code snippet), for which **a message is added to the queue**, and when the event-loop picks it up, the call stack **starts stacking frames** for `baz()`, `bar()`, and `foo()` at the relevant points of execution.

Once the execution of the functions is complete one by one, their frames are **removed from the call stack**, while the message is **still waiting in the queue**, until `baz()` is popped from the stack.

Remember, the function calls are **never direct in JavaScript**, they're done **via messages**. So whenever you hear someone say that JavaScript itself is an asynchronous programming language, assume that they are talking about its built-in "answering machine", and how you're free to leave messages.

But what about the specific asynchronous methods?

So far I've not touched on APIs such as `setTimeout()` and AJAX, those are the ones that are **specifically referred to as asynchronous**. Why is that?

It's important to understand what exactly is being synchronous or asynchronous. JavaScript, with the help of events and the event-loop, may practice **asynchronous processing of messages**, but that **doesn't mean *everything* in JavaScript is asynchronous**.

Remember, I told you the message didn't leave until the call stack was **empty of its corresponding frames**, just like you didn't leave for a movie until you got your answer — that's **being synchronous**, you are there waiting **until the task is complete**, and you get the answer.

Waiting **isn't ideal in all scenarios**. What if after leaving a message, instead of waiting, you can leave for the movie? What if a function can retire (emptying the call stack), and its message can be dequeued even before the task of the function is complete? What if you can have code executed asynchronously?

This is where APIs such as `setTimeout()` and AJAX come into the picture, and what they do is… hold on, I can't explain this without going back to Mr X, which we'll see in the second part of this article. Stay tuned.

Report a bug

2 Comments Sort by Newest

Add a comment...

Reshia Minggia · Full.time victim at I am a victim
Beautiful judicial stanza

Please institute justice

Like · Reply · Nov 15, 2016 9:06am

Dj3dw
I am tuned. Where is part 2, I'm on the edge of my seat...

Like · Reply · Nov 11, 2016 10:03pm

Facebook Comments Plugin

Report a bug