Hello, my name is Olga Naranovich. My presentation is devoted to Event Loop, but before we talk about the event loop we need to understand how the JS’s engine works, what happens when JavaScript is executed and what role the event loop plays in JS’s execution.

Javascript is capable of Asynchronous operations through the single thread of execution. This means that an event such as a database request can be performed, and while the request is being performed other events can still be performed such as a user scrolling or entering data. This is unlike most other languages. They usually require the callback to return with its response before executing any following code.

We know that JS is a single-threaded programming language. So if it has one thread, that means it has a single call stack and can do one thing at a time.

Widely known JS engine is the V8 from Google. The engine v8 or the other ones in the other browsers basically is entirely made up of two pieces: Heap and Callstack.

Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory. Function calls form a stack of frames.

More we can see in the following example.

Every time a function needs to be executed in JavaScript, it’s added to the stack. If that function calls another function within it, then that function is added to the top of the stack. When the function has finished its execution it’s removed from the Callstack. And so on until the stack is empty. So the browser engine executes JavaScript in one thread. It cannot pause the event processing, switch to another event, and then resume the execution of the first one. All events are processed sequentially and each - to the bitter end.

But what if we make a request or put a timeout on something or have any other function that needs a lot of time to be executed?

In theory that should freeze the entire browser until it’s executed. This means that the browser will not be able to display anything on the screen, or execute any other code. It just will stop.

The solution is to use asynchronous callback functions. But how is asynchrony possible in a single-threaded language?

The short answer is that the asynchronous behavior is not a part of the JavaScript language itself. Rather they are built on top of the core JavaScript language in the browser, when there’re web APIs, a callback queue, and an event loop. So, if we make a request or put a timeout on something or work with DOM events we do it via callback functions. Every time we call functions mentioned above  — they’re added to the callback queue.

A queue is a list of messages to process. Each message has an associated function that is called to process this message. But you somehow need to send them to the Callstack. The Event loop is responsible for this.

Event loop is a constantly running process that checks if the Callstack is empty. If there is something in the callback queue that is waiting, it is moved to the Callstack. If not, then nothing happens.

Let's have a look at the execution of such code in the browser. So, we run the console.log code and immediately see the result in the console. The next function is setTimeout, what happens to it? We pass this function with callback, its name is cb, the second argument is the delay value. SetTimeout is what we are provided by the browser, it’s not in the JS engine. At this moment the browser starts the timer, and the countdown is actually performed in parallel to the engine. Therefore, the stack can remove the execution of this function to call the next one. This is what this console.log is. Function is executed and its result is displayed in the console.

And now with a clean callstack, we have something running in API. When the API ends its work in 5s, the callback moves to the callback queue. And now it's time for the event loop. As soon as a task appears in the queue, it checks if the stack is clean. And if it is empty, it moves the first task in the queue there. The task is immediately executed.

Let's have a look at some more examples when the event loop is used. This example differs from the previous one only because it has the delay argument as 0. This technique is used when you need to schedule something right after the execution of the main code.

This is a general scheme for all asynchronous requests. For example, AJAX requests or processing DOM events work exactly in the same way. In this example we will receive a XHR response after the execution of the main code.

And for example, this code is from this presentation. It starts the animation of the icon event loop under certain conditions using setTimeout and setInterval.

As a conclusion, it can be said that it is important to understand the internal features of the implementation of asynchronous operations in JavaScript and actively use them in writing programs. After all, the main task of any developer is to create a convenient, understandable and, what is most important, fast user interface.