Chap 11 and 18 Notes

Synchronous Programming Model - Things happen one at a time

Asynchronous Programming Model - Multiple things happen at the same time

The setTimeout function holds a function from starting depending on the amount of time its told. I can see this being useful on programs where an object needs to load first before the next object does. An example would be waiting to display text until a picture if fully displayed.

I'm having a hard time understanding the code about looking up a cache in storage:

import {bigOak} from " . /crow-tech" , 
bigOak. readStorage(" food caches" , 
Let firstCache 
caches ; 
bigOak . readStorageCfirstCache , 
console. Log(info) ; 
caches => { 
info => { 

By the looks of it, this looks like it's a function within a function. bigOak.readStorage is being called within another bigOak.readStorage function

Request-response pairs - A message is sent and immediately gets a reply with a confirmation receipt and a reply from a question asked in the original message if given

**Callbacks**

A **send** method expects *the name of the target nest*, *the type of the request,* and *the content of the request* as its first three arguments. It also expects *a function to call when a response comes in* as the fourth argument.

bigOak. Pasture", "note" , 
"Let's caw loudly at 7PM" , 
( ) => delivered. '1)); 

\*\*\* The request type has to be defined. In the above code, it is "note" \*\*\* which is what you see as code underneath. The code below needs to be written so that the request type "note" is defined.

import {defineRequestType} from 
defineRequestType("note" , 
(nest , 
" . /crow-tech" ; 
content, source, done) 
console. ${nest . name} received note: ${content}'); 
done() ; 

**defineTypeRequest** function above defines a new type of request

**From the book:**

The fourth argument given to the handler, done, is a callback function that it must call when it is done with the request. If we had used the handler’s return value as the response value, that would mean that a request handler can’t itself perform asynchronous actions. A function doing asynchronous work typically returns before the work is done, having arranged for a callback to be called when it completes. So we need some asynchronous mechanism—in this case, another callback function—to signal when a response is available.

**Promises**

A **Promise** is an asynchronous action that may complete at some point and produce a value.

To create a promise, use **Promise.resolve** function that ensures that the value you give it is wrapped in a promise.

let fifteen 
Promise . resolve(15) ; 
fifteen . then(value => consoLe.log('Got ${vaLue}' )); 
* Got 15 
To get the result of a promise, you can use its then method. This registers a 
callback function to be called when the promise resolves and produces a value. 
You can add multiple callbacks to a single promise, and they will be called, 
even if you add them after the promise has already resolved (finished). 
But that's not all the then method does. It returns another promise, which 
resolves to the value that the handler function returns or, if that returns a 
promise, waits for that promise and then resolves to its result. 

Chap 18 - HTTP and Forms

**Status Codes:**

2xx - Succeed

4xx - Request Error

5xx - Server Error

Example form

1 <form method—" GET" action '[exampLe/message . html 
<p>Name : <br><input type= "text" name="name"></p> 
<p>Message : name="message"></textarea></p> 
<p><button type—I' submit 
Name : 
Message: 
Send 

GET requests should be used for requests that do not have side effects but simply ask for information. Requests that change something on the server, for example creating a new account or posting a message, should be expressed with other methods, such as POST. Client-side software such as a browser knows that it shouldn’t blindly make POST requests but will often implicitly make GET requests—for example to prefetch a resource it believes the user will soon need.

**Fetch**

Machine generated alternative text:
To get at the actual content of a response, you can use its text method. 
Because the initial promise is resolved as soon as the response's headers have 
been received and because reading the response body might take a while 
longer, this again returns a promise. 
fetch( exampLe/data . txt " ) 
. then(resp => resp. text()) 
. then(text => console. Log(text)); 
// * This is the content of data. txt 
A similar method, called j son, returns a promise that resolves to the value you 
get when parsing the body as JSON or rejects if it's not valid JSON. 

**Form Fields**

Machine generated alternative text:
text 
password 
checkbox 
radio 
file 
A single-line text field 
Same as text but hides the text that is typed 
An on/off switch 
(Part of) a multiple-choice field 
Allows the user to choose a file from their computer 
Form fields do not necessarily have to appear in a tag. You can put 
them anywhere in a page. Such form-less fields cannot be submitted (only a 
form as a whole can), but when responding to input with JavaScript, we often 
don't want to submit our fields normally anyway. 
4 
<p><input 
<p><input 
<p><input 
<p><input 
<input 
<input 
<p><input 
type 
type 
type 
type 
type 
type 
type 
•J (checkbox) 
(radi o) 
Choose File No file chosen 
"text" vaLue—" abc"> 
"password" vaLue—" abc (password)</p> 
'l checkbox" checked> (checkbox)</p> 
"radio" name="choice"> 
"radio" name—I' choice" checked> 
"radio" name—"choice"> (radio)</p> 
(password) 
(file) 

**Storing data client-side**

Machine generated alternative text:
The Local Storage object can be used to store data in a way that survives page 
reloads. This object allows you to file string values under names. 
locaLStorage . setltem( '[username" , "marijn"); 
console . Log(LocalStorage . getltem( " username " ) ) ; 
3 // * marijn 
local Storage . removeltem( '"username " ) ; 
mar 1] rl 
A value in Local Storage sticks around until it is overwritten, it is removed 
with removeltem, or the user clears their local data. 

**\*\*\*This code implements a crude note-taking application. It keeps a set of named notes and allows the user to edit notes and create new ones\*\*\***

Notes: <select></select> <button>Add</button><br>

<textarea style="width: 100%"></textarea>

<script>

let list = document.querySelector("select");

let note = document.querySelector("textarea");

let state;

function setState(newState) {

list.textContent = "";

for (let name of Object.keys(newState.notes)) {

let option = document.createElement("option");

option.textContent = name;

if (newState.selected == name) option.selected = true;

list.appendChild(option);

}

note.value = newState.notes[newState.selected];

localStorage.setItem("Notes", JSON.stringify(newState));

state = newState;

}

setState(JSON.parse(localStorage.getItem("Notes")) || {

notes: {"shopping list": "Carrots\nRaisins"},

selected: "shopping list"

});

list.addEventListener("change", () => {

setState({notes: state.notes, selected: list.value});

});

note.addEventListener("change", () => {

setState({

notes: Object.assign({}, state.notes,

{[state.selected]: note.value}),

selected: state.selected

});

});

document.querySelector("button")

.addEventListener("click", () => {

let name = prompt("Note name");

if (name) setState({

notes: Object.assign({}, state.notes, {[name]: ""}),

selected: name

});

});

</script>

Machine generated alternative text:
The script gets its starting state from the "Notes" value stored in 
local Storage or, if that is missing, creates an example state that has only a 
shopping list in it. Reading a field that does not exist from local Storage will 
yield null . Passing null to JSON . parse will make it parse the string "nuLL " 
and return null . Thus, the I I operator can be used to provide a default value 
in a situation like this. 
The setState method makes sure the DOM is showing a given state and 
stores the new state to Local Storage. Event handlers call this function to 
move to a new state. 
The use of Obj ect. assign in the example is intended to create a new object 
that is a clone of the old state . notes , but with one property added or 
overwritten. Obj ect . assign takes its first argument and adds all properties 
from any further arguments to it. Thus, giving it an empty object will cause it to 
fill a fresh object. The square brackets notation in the third argument is used to 
create a property whose name is based on some dynamic value. 
There is another object, similar to Local Storage, called sessionStorage. 
The difference between the two is that the content of sessionStorage is 
forgotten at the end of each session, which for most browsers means whenever 
the browser is closed. 

Function Constructor (from Chap 10) :

let plusOne = Function("n", "return n + 1;");

console.log(plusOne(4));

// → 5