WDD 330: Wk 3 Readings

**10 Mobile UX Design Principles**

basic usability principles of interface design:

-Learnability: How easily first-time users complete basic tasks.

-Efficiency: How quickly users perform basic tasks once they’ve learned the design.

-Memorability: The ability of users to remember how to use the system.

-Errors: The amount and severity of errors users make and how easily they can correct them.

-Satisfaction: How pleasant the experience of using the design was.

*10 design principles key to creating great mobile user experiences*:

1. Content Prioritization

-Attention span 8 secs

-Less is more when designing

-essential content only

-icons instead of text when possible

1. Make Navigation Intuitive

-“If the user can’t use it, it doesn’t work.” – Susan Dray

1. Touchscreen Target Sizes

-Apple iPhone Human Interface Guideline’s min target size 44 pixels wide x 44 pixels tall

Windows Phone UI Design and Interaction Guidelines target size 9mm/34px w/min target 7mm/26px

-spacing between targets important too

-include an undo button for inevitable human error

1. Provide User Control

-Allow users to make decisions

-provide warnings

-allow them to demo or continue as a guest before committing

1. Legible Text Content

-min 16 pixels (or 11 points)

-30-40 characters per line for mobile

-spacing and layout improve legibility

1. Make Interface Elements Clearly Visible

-contrast between content and background in design so it’s legible

-WC3’s Web Content Accessibility Guidelines

1. Hand Position Controls

-49% of people rely on one thumb- right and left thumbs should be considered

1. Minimize Data Input

-provide autocomplete, shorten forms, remove unnecessary fields, use “remember me” options

1. Create a Seamless Experience

-reduce friction, minimize steps and page loads

-make content accessible even w/o online connection, provide alternative paths

-use mobile phone features: camera to scan barcodes, GPS for identifying locations and touch ID in place of passcodes

-synchronization across devices is a key priority for creating seamless experiences.

1. Test Your Design

-test early, often, continuously and optimize

-test w/ real users

**Introduction to web API’s**

*APIs in client-side JavaScript*

-browser API’s – built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it.

-3rd-party API’s – you generally have to retrieve their code and information from somewhere on the Web ie. Twitter API

*What can API’s do?*

-MDN API’s index page

-manuipulate DOM

-fetch data from the server

-drawing and manipulation graphics

-Audio and Video API’s

-device APIs

-client-side storage APIs

// play/pause audio

playBtn.addEventListener('click', () => {

// check if context is in suspended state (autoplay policy)

if (audioCtx.state === 'suspended') {

audioCtx.resume();

}

// if track is stopped, play it

if (playBtn.getAttribute('class') === 'paused') {

audioElement.play();

playBtn.setAttribute('class', 'playing');

playBtn.textContent = 'Pause'

// if track is playing, stop it

} else if (playBtn.getAttribute('class') === 'playing') {

audioElement.pause();

playBtn.setAttribute('class', 'paused');

playBtn.textContent = 'Play';

}

});

// if track ends

audioElement.addEventListener('ended', () => {

playBtn.setAttribute('class', 'paused');

playBtn.textContent = 'Play'

});

*Scheduling: setTimeout and setInterval*

setTimeout allows us to run a function once after the interval of time.

setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

For instance, this code calls sayHi() after one second:

*setTimeout*

function sayHi() {

alert('Hello');

}

setTimeout(sayHi, 1000);

With arguments:

function sayHi(phrase, who) {

alert( phrase + ', ' + who );

}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John

If the first argument is a string, then JavaScript creates a function from it.

So, this will also work:

setTimeout("alert('Hello')", 1000);

But using strings is not recommended, use arrow functions instead of them, like this:

setTimeout(() => alert('Hello'), 1000);

Pass a function, but don’t run it

Novice developers sometimes make a mistake by adding brackets () after the function:

// wrong!

setTimeout(sayHi(), 1000);

That doesn’t work, because setTimeout expects a reference to a function. And here sayHi() runs the function, and the result of its execution is passed to setTimeout. In our case the result of sayHi() is undefined (the function returns nothing), so nothing is scheduled.

*clearTimeout*

-timers section of HTML Living Standard

let timerId = setTimeout(...);

clearTimeout(timerId);

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

let timerId = setTimeout(() => alert("never happens"), 1000);

alert(timerId); // timer identifier

clearTimeout(timerId);

alert(timerId); // same identifier (doesn't become null after canceling)

*setInterval*

let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

// repeat with the interval of 2 seconds

let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

Time goes on while alert is shown

In most browsers, including Chrome and Firefox the internal timer continues “ticking” while showing alert/confirm/prompt.

So if you run the code above and don’t dismiss the alert window for some time, then the next alert will be shown immediately as you do it. The actual interval between alerts will be shorter than 2 seconds.

Zero delay scheduling with setTimeout(func, 0) (the same as setTimeout(func)) is used to schedule the call “as soon as possible, but after the current script is complete”.

The browser limits the minimal delay for five or more nested calls of setTimeout or for setInterval (after 5th call) to 4ms. That’s for historical reasons.

Please note that all scheduling methods do not guarantee the exact delay.

For example, the in-browser timer may slow down for a lot of reasons:

The CPU is overloaded.

The browser tab is in the background mode.

The laptop is on battery saving mode.

\*\*\*I don’t have any questions. I know this is supposed to be review, but most of this I don’t have embedded in my brain, so it’s a lot to take in again. At least the concepts are not entirely new, but I will need to refer to these notes for reference.