

# Asynchronous

- Matching human perception and expectations
- Asynchronous execution
- Fetch API
- Worker thread

# Responsive\* User Interfaces

- A responsive UI *feels* like it responds in a timely manner
  - Examples:
  
- This is accomplished primarily in **two ways**:
  1. Designing for human perception and expectations
  2. Using asynchronous execution

**Responsiveness is not just system performance**

\* not related to **responsive layouts** (the layout term for adapting to different window sizes and/or devices)

# Human Perception of Time

- Knowing *the duration of perceptual and cognitive processes* can inform the design of interactive systems that feel responsive
- Can examine results of *Mental Chronometry* studies
  - Minimal time to detect a gap of silence in sound: 4 ms
  - Minimal time to be affected by a visual stimulus: 10 ms
  - Time that vision is suppressed during a saccade: 100 ms
  - Maximum interval between cause-effect events: 140 ms
  - Time to comprehend a printed word: 150 ms
  - Visual-motor reaction time to an observed event: 1 s
  - Time to prepare for conscious cognition task: 10 s
  - Duration of unbroken attention to a single task: 6 s to 30 s

***(times approximate)***

# Continuous Latency

**Minimal time to be affected by a visual stimulus: 10 ms**

→ continuous input latency should be less than 10ms

e.g. dragging a shape, how far behind the cursor is the shape

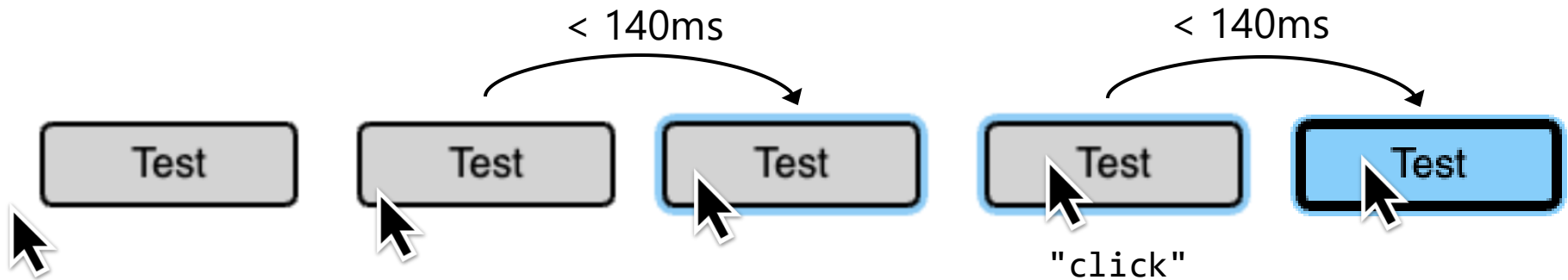


# Input Feedback

**Maximum interval between cause-effect events: 140 ms**

→ input feedback should appear in less 140ms

e.g. time between pressing a button until the feedback changes





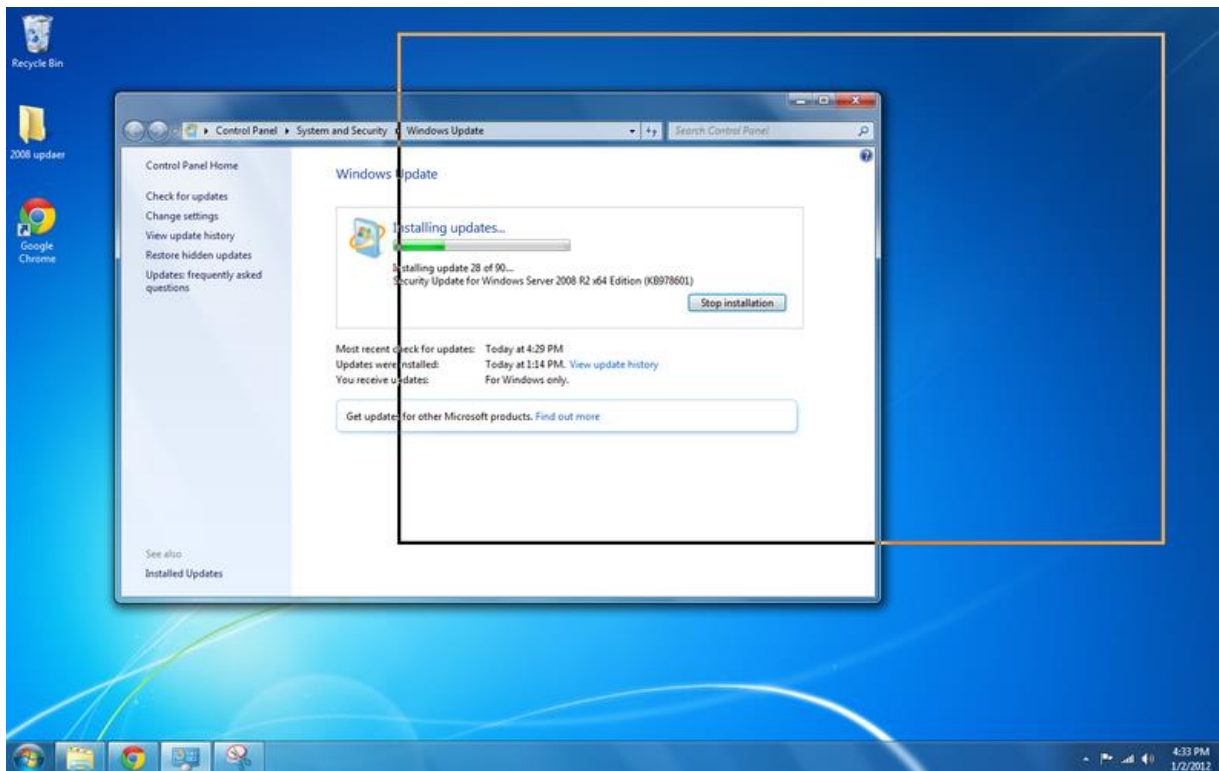
User Perception of Latency & Latency Improvements in Direct and Indirect Touch  
- [https://youtu.be/1dKIMZrM\\_sw](https://youtu.be/1dKIMZrM_sw)

# Graceful Degradation of Feedback

Simplify feedback for high-computation continuous input tasks

- *Examples*

- window manager updates window rendering after drag
- graphics editor only draws object outlines during manipulation
- CAD package reduces render quality when panning or zooming



# Busy Indicators

## Visual-motor reaction time to an observed event: 1 s

→ Display *busy indicators* for operations that take 1s to about 3-4s

- Busy indicator design
  - Use visually cohesive cyclic animations (not repeating “progress” indicators)



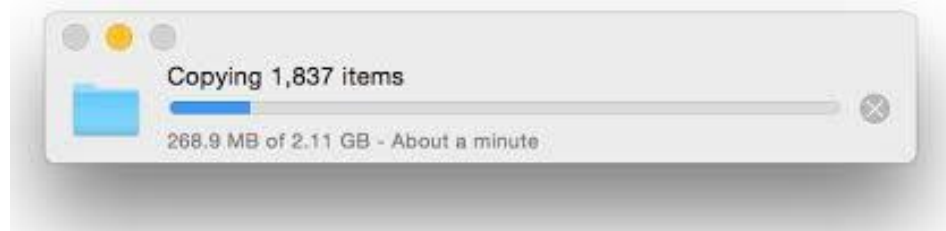


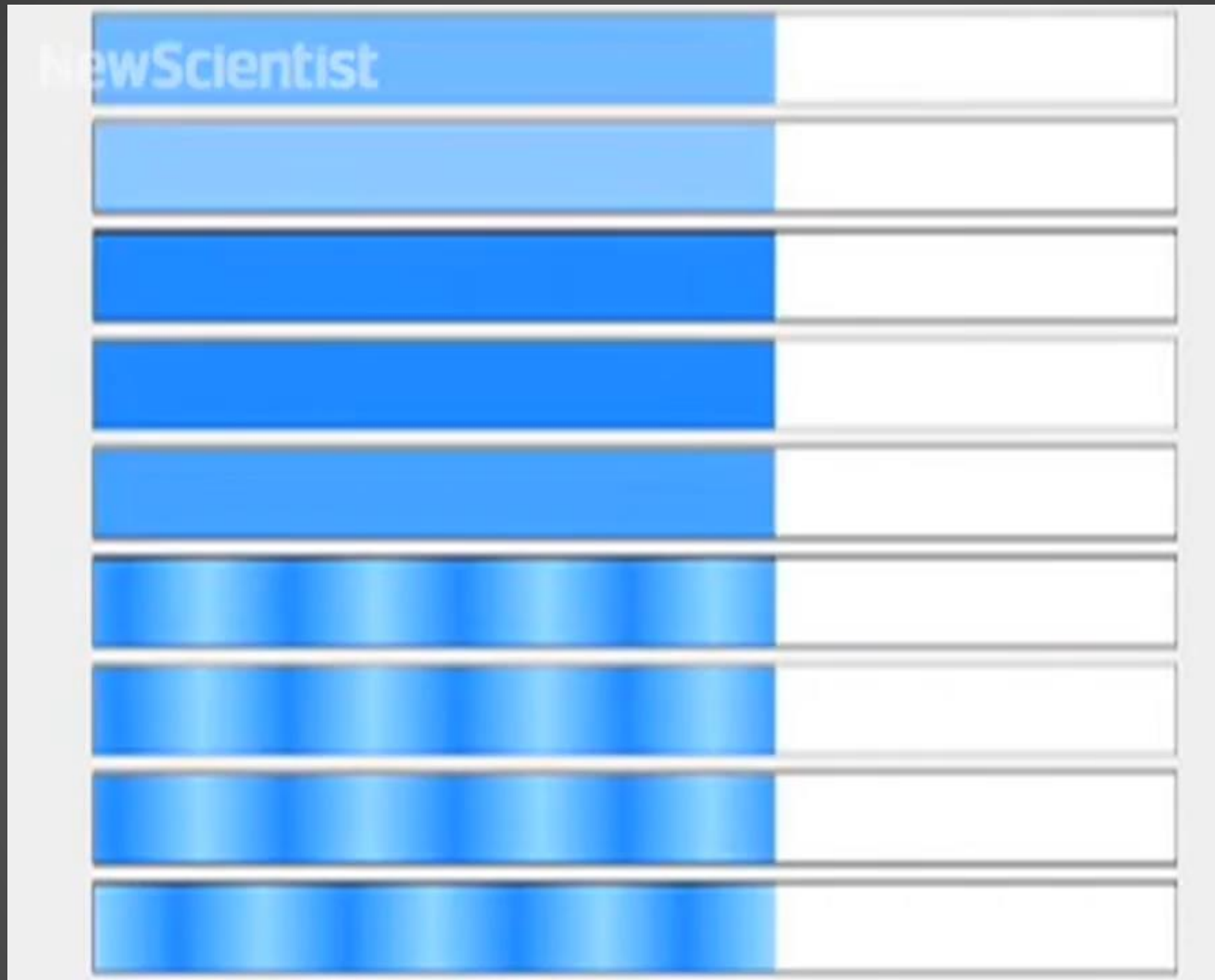
# Progress Bars

## Visual-motor reaction time to an observed event: 1 s

→ Display *progress bars* for operations more than 3-4s

- Progress bar design
  - Show work remaining, not work completed
  - Use human precision, not computer precision  
(Bad: “243.5 seconds remaining”, Good: “about 4 minutes”)
  - Show smooth progress, not erratic bursts
  - Show total progress when multiple steps, not only step progress
  - Display finished state (e.g. 100%) very briefly at the end





Harrison et al. Faster Progress Bars (2010)

- <https://www.newscientist.com/article/dn18754-visual-tricks-can-make-downloads-seem-quicker/>

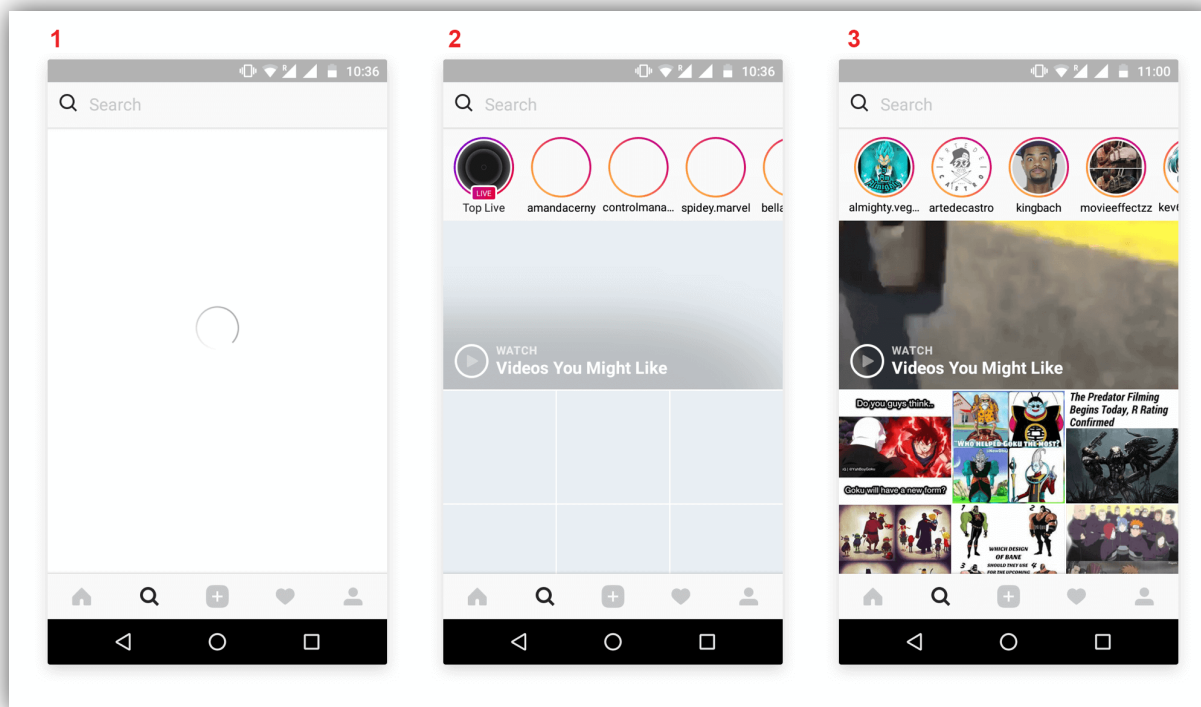
# Progressive Loading

## Visual-motor reaction time to an observed event: 1 s

→ Use *skeleton placeholders* when loading takes more than 1 s

- Advantages:

- User adjusts to a layout they'll eventually see
- Loading process seems faster because there is an initial results

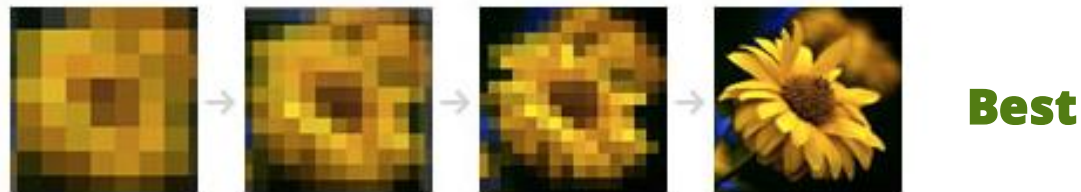


Instagram minimal version skeleton screen

<https://blog.iamsuleiman.com/stop-using-loading-spinner-theres-something-better/>

# Progressive Loading

- Provide user with *some data* while loading rest of data
- *Examples*
  - word processor shows first page as soon as document opens
  - search function displays some items as soon as it finds them
  - webpage displays low resolution images, then higher resolution



# Responsiveness by Predicting Next Operation

- Use periods of low load to pre-compute responses to high probability requests. Speeds up subsequent responses.
- *Examples*
  - text search function looks for next occurrence of the target word while user looks at the current
  - web browser “prerenders” pages that are likely to be visited next

# Progressive Loading

Time to prepare for conscious cognition task: 10 s

→ Display image of document on last save,  
while real one loads in less than 10s

**Meet The Canon Cat.**  
A new breed of office machine.

The Canon Cat is the world's first Work Processor. It is a simple but powerful office machine.

The Canon Cat is not a typewriter, electronic or otherwise. Yet minutes after you plug it in you can be typing on The Cat like a veteran.

The Canon Cat is not a word processor. Yet it will let you write and edit as well as the most sophisticated word processor.

The Canon Cat is not a personal computer. Yet it will let you do calculations right in the text, store information and communicate with other office machines.

What is The Cat? As we said, it's the world's first Work Processor. It can help you write and edit, communicate and calculate. It'll even dial your phone.

**The great leap forward.**

The Canon Cat is the brainchild of the man who originated the Macintosh Computer. This time he wanted to make, not a computer, but an office appliance so simple that anyone could plug it in and use it—like a toaster.

He analyzed how people (particularly office people) think and work. Then he designed The Cat to work the way people think. Which makes work easier.

For example, if you look over the keyboard on this page you'll see that it is just like a traditional typewriter keyboard. No fancy computer keys with confusing names like Access and Control. It's familiar and easy to use.

See these rosy Leap keys? They are The Cat's most fascinating feature. Just press one down, type in a few letters you remember from a document—

and you're where you want to be—instantly. No menu. No files. No mouse.

**A most productive pet.**

If you're ready to move up from typewriters to the world of microchips and screens, The Canon Cat is for you.

It's from Canon, home of a long tradition of office innovations, including personal copiers and desktop laser beam printers. The Cat has been designed to work especially well with Canon Printers including The Cat180 Daisy Wheel Printer and the Canon Laser Beam Printer.

The Cat is so easy to learn that you or your employees won't have to disappear for days into often frustrating training sessions.

Anyone can become an expert on The Cat in just a few hours.

The Cat is most affordable.

The Canon Cat will make mountains of work disappear faster and easier than ever before.

That's why we call it the world's first Work Processor.

**Canon Cat**  
The Advanced WORK Processor

Macintosh is a trademark of Apple Computer, Inc.

# Asynchronous Execution

- Execute tasks independently from the main program flow
  - “Do more than one thing at once”
- Two main types of Asynchronous Execution
  1. **Asynchronous Programming**
  2. **Threading**

# Asynchronous Programming

- A paradigm that allows for execution of tasks in a non-blocking manner in a **single thread**
  - NOT concurrent execution
- Related concepts in JavaScript and other Languages
  - Event driven programming
  - Promise/Future
  - Coroutines
  - Non-blocking I/O



The diagram illustrates the JavaScript Event Loop mechanism. It features several components:


- Call Stack:** A vertical container on the left, outlined in orange, representing the current execution context.
- Web APIs:** A central box outlined in purple containing various browser APIs such as `fetch`, `setTimeout`, `URL`, `localStorage`, `sessionStorage`, `HTMLDivElement`, `document`, `IndexedDB`, and `XMLHttpRequest`. A note below says "Many more...".
- Task Queue:** A horizontal container outlined in pink, representing the queue of tasks that are ready to be executed.
- Microtask Queue:** A horizontal container outlined in green, representing the queue of microtasks that are processed before the main task queue.
- Event Loop:** A vertical container on the bottom left outlined in blue, containing a circular arrow icon, representing the loop that checks and processes the queues.

JS JavaScript Visualized

# Event Loop

Web APIs, Task Queue, Microtask Queue

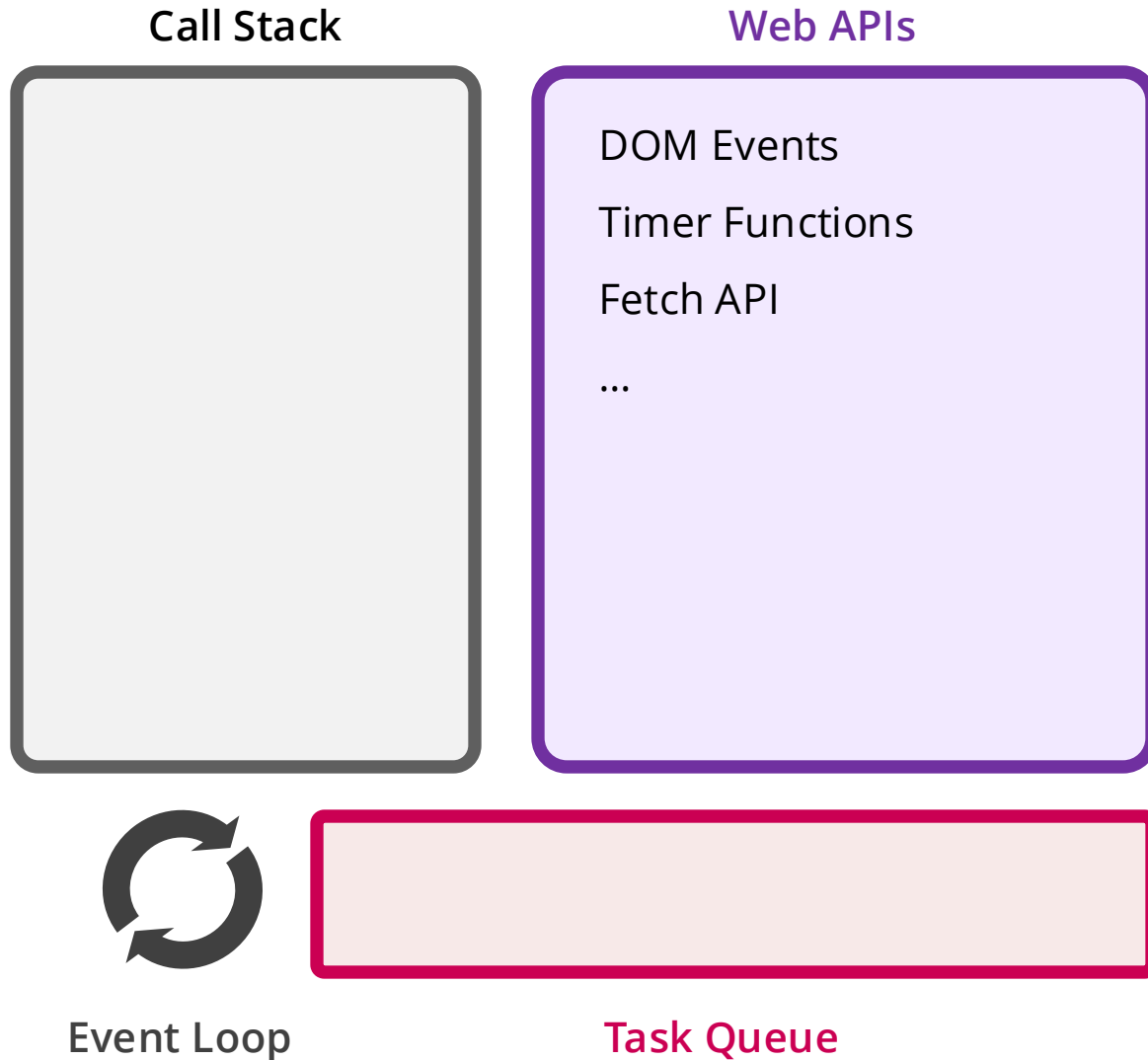
```
1 Promise.resolve().then(() => {
2   console.log("what")
3 })
4
5 setTimeout(() => {
6   console.log("order?")
7 }, 100);
8
9 console.log("In")
```



JavaScript Visualized - Event Loop, Web APIs, (Micro)task Queue, **Lydia Hallie**

- <https://www.youtube.com/watch?v=eiC58R16hb8>

# (Simplified) JavaScript Runtime Environment



# Call Stack

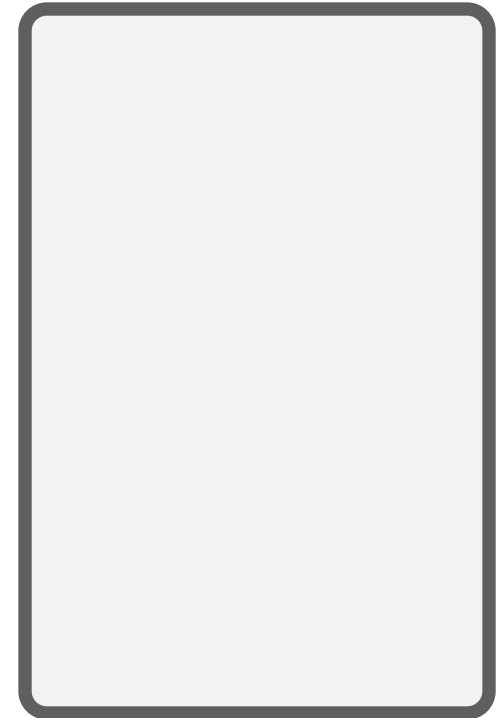
```
1 console.log("start");
2
3 function bar() {
4   console.log("bar");
5 }
6
7 function foo() {
8   bar();
9   console.log("foo");
10 }
11
12 foo();
13
14 console.log("end");
15
```

Call Stack



at line 12

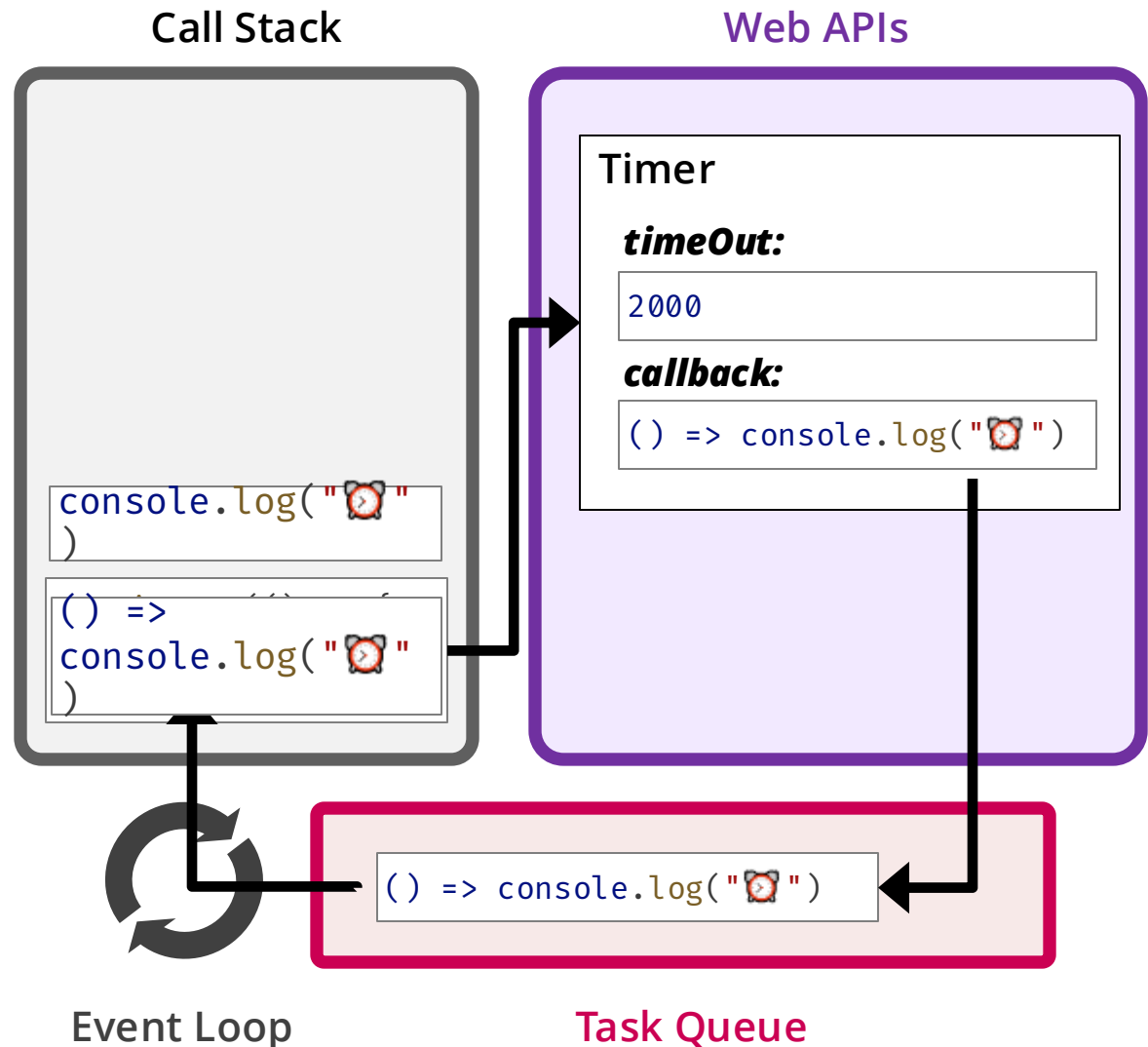
Call Stack



at line 15

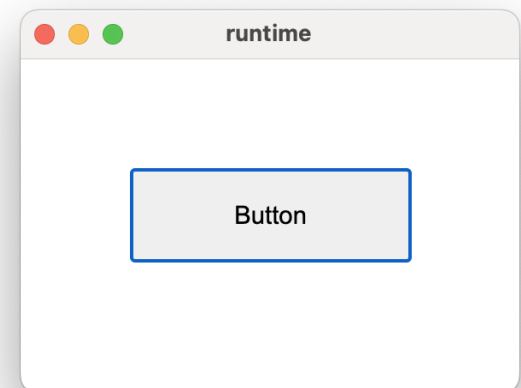
# Web APIs, Task Queue, Event Loop

```
1 console.log("start");
2
3 setTimeout(() => {
4   console.log("🕒");
5 }, 2000);
6
7 function bar() {
8   console.log("bar");
9 }
10
11 function foo() {
12   bar();
13   console.log("foo");
14 }
15
16 foo();
17
18 console.log("end");
19
```



# runtime

- Walkthrough for runtime environment
- Demos
  1. What if timer is 0ms?
  2. Uncomment `long()` in main
  3. Uncomment `long()` in button callback



# Callbacks

- **Input events** are **asynchronous methods**

- We handle them as callbacks bound to a DOM element

```
button.addEventListener("click", () => {  
  console.log("💣 button");  
  // do something
```

```
});
```

```
1 asyncOperation1(function(result1) {  
2   · asyncOperation2(result1, function(result2) {  
3     · · · asyncOperation3(result2, function(result3) {  
4       · · · · · // More nested callbacks ...  
5       · · · · · });  
6     · · });  
7   });  
8 };
```

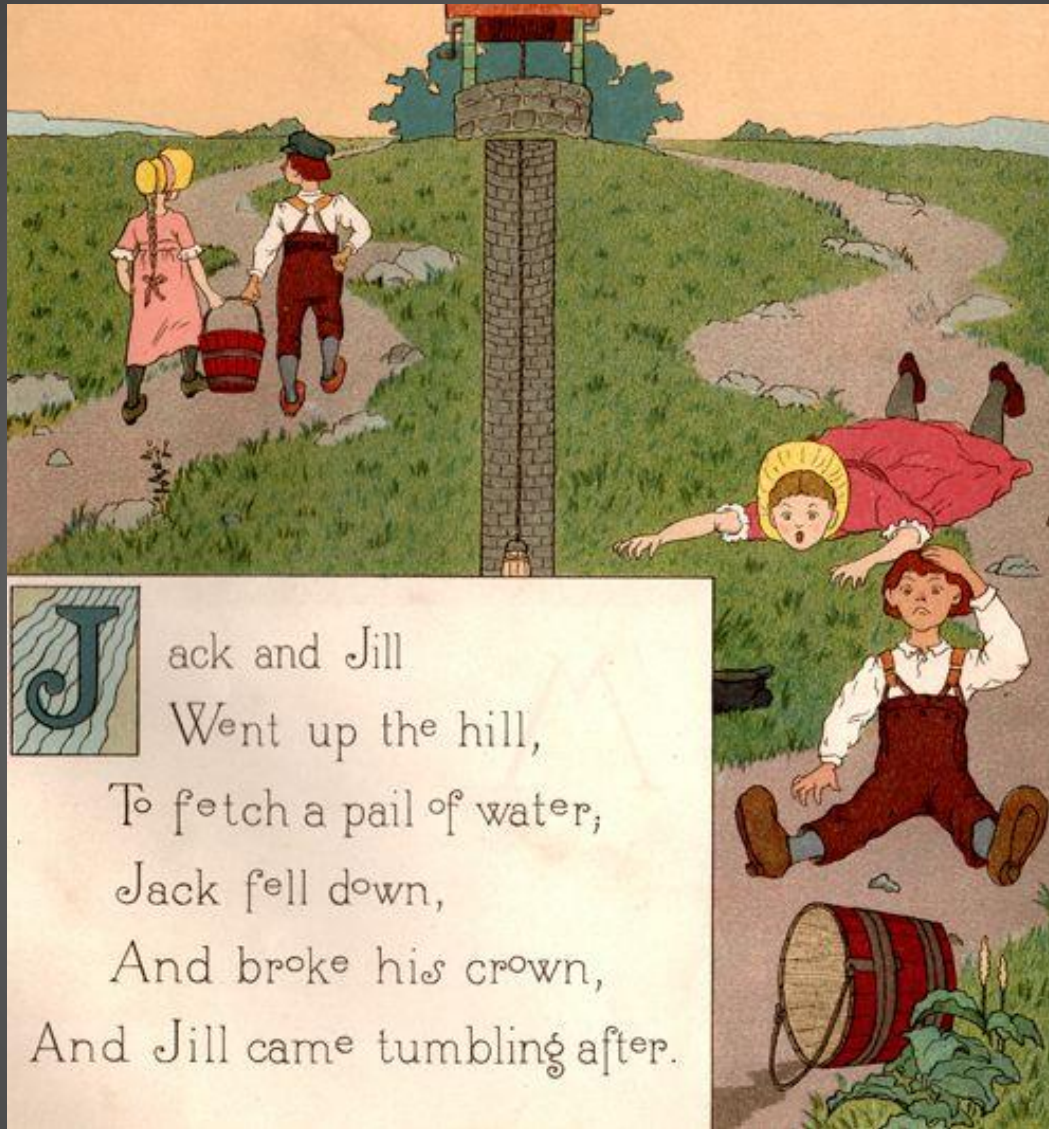
## Callback Hell

- [https://medium.com/@raihan\\_tazdid/callback-hell-in-javascript-all-you-need-to-know-296f7f5d3c1](https://medium.com/@raihan_tazdid/callback-hell-in-javascript-all-you-need-to-know-296f7f5d3c1)

# Fetch API

- An interface for fetching resources across the network
- `fetch()` function
  - starts the process of fetching a resource from the network
- Returns a “Promise” object with three states:
  - *Pending*, when fetch process is happening
  - *Resolved*, when the process was successful and there’s a valid response
  - *Rejected*, when the process failed and there’s an error






**J**ack and Jill  
Went up the hill,  
To fetch a pail of water,  
Jack fell down,  
And broke his crown,  
And Jill came tumbling after.

## Jack and Jill Nursery Rhyme Analogy of Promises

- Inspired by <https://blog.greenroots.info/javascript-promises-explain-like-i-am-five>

# fetch

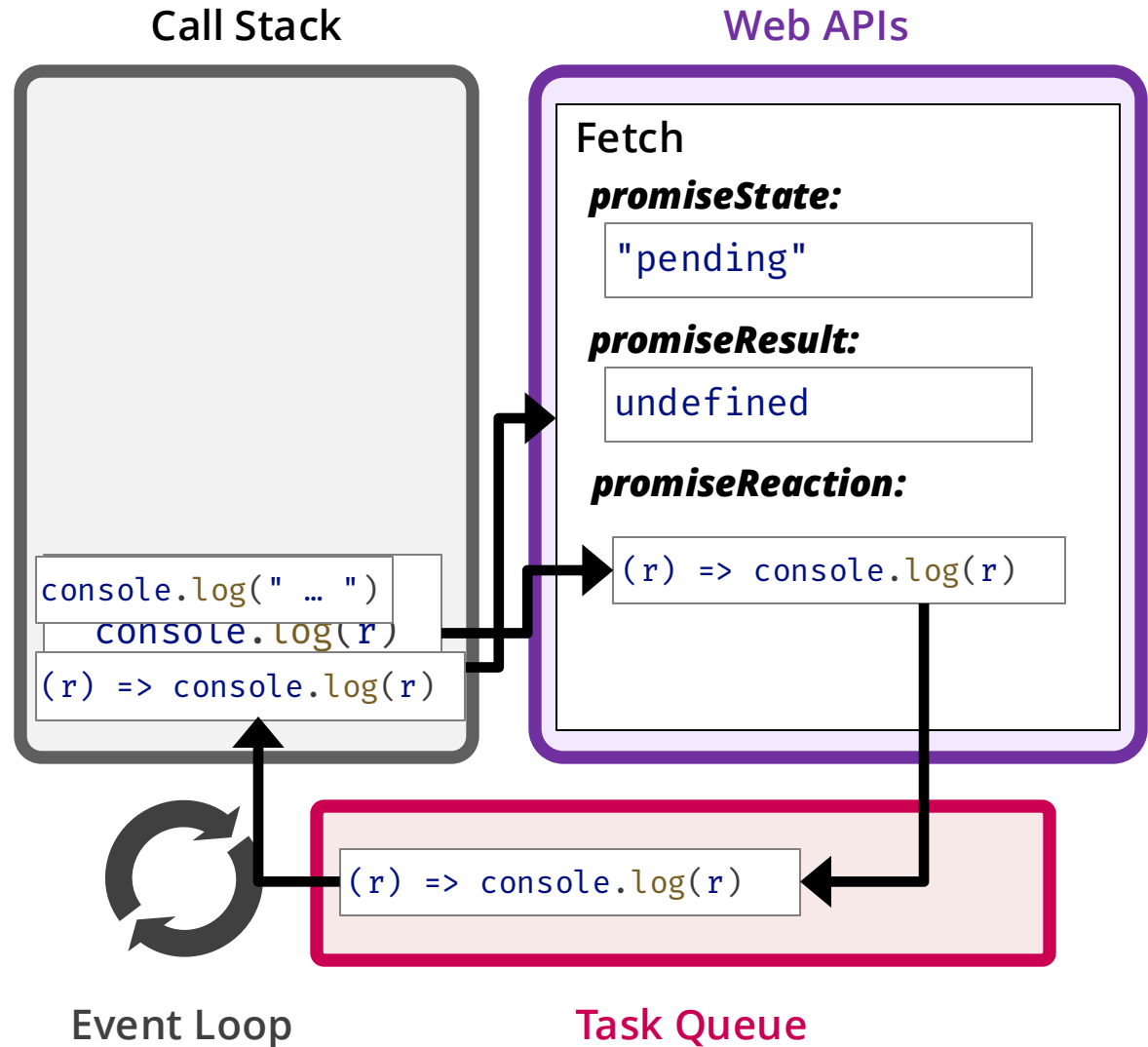
- Demos:
  - async function
  - Network throttling to simulate slow connection
- doFetch1() using Promises
  - fetch() with chained .then( ... )
  - error handling
- doFetch2() with async/await



```
fetch
https://pokeapi.co/api/v2/pokemon?limit=100000&offset=0 Fetch
{
  "count": 1292,
  "next": null,
  "previous": null,
  "results": [
    {
      "name": "bulbasaur",
      "url": "https://pokeapi.co/api/v2/pokemon/1/"
    },
    {
```

# JavaScript Runtime with Fetch API

```
1 fetch(url)
2   .then((r) => {
3     console.log(r);
4   })
5   .catch((e) => {
6     console.error(e);
7   });
8
9
10
11
12
13
14
15
16
17
18
19
```



# Fetch Progress

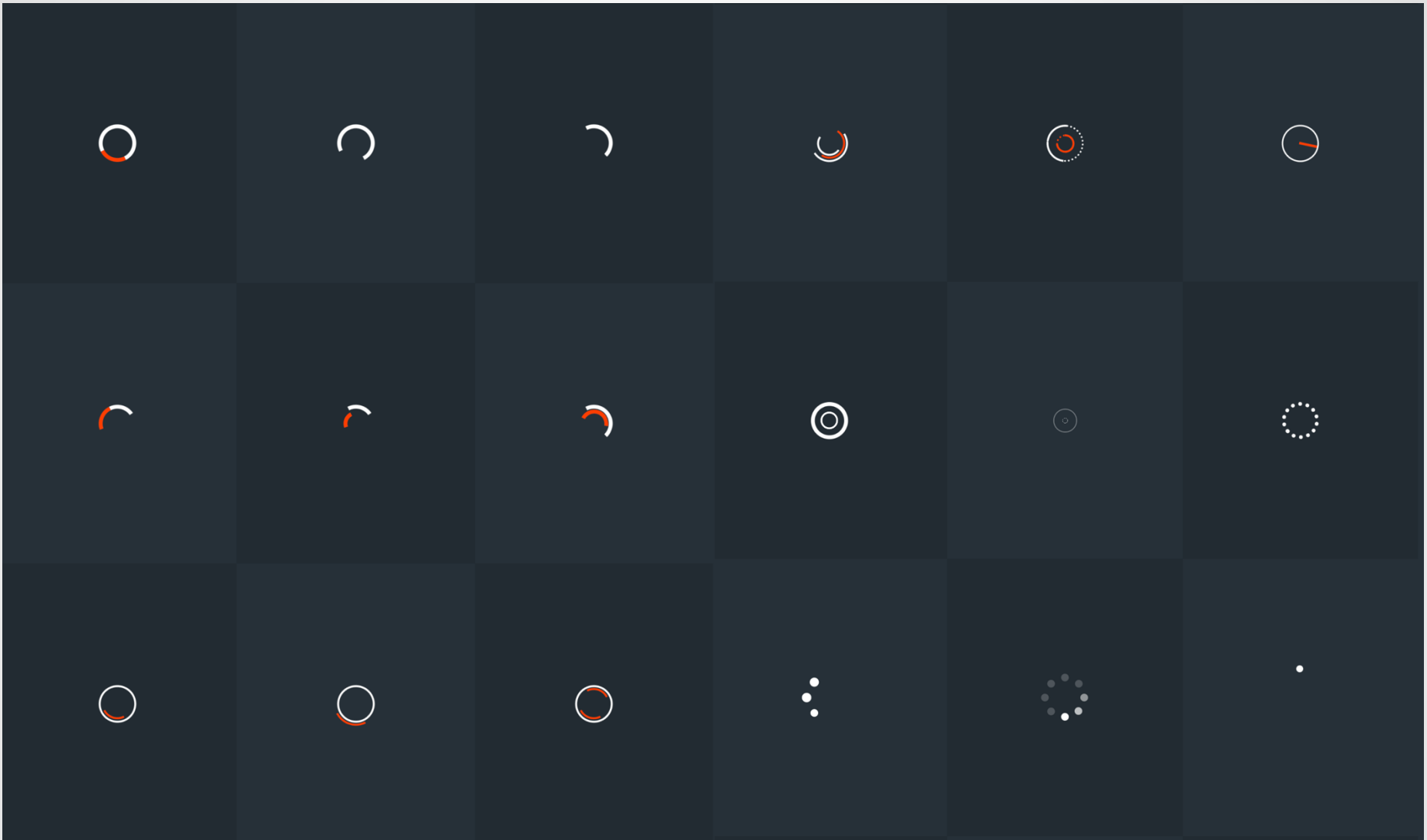
- Surprisingly complex to get progress during fetch
  - Use `ReadableStream`, read in chunks, ...
- Link below shows an approach

## fetch (with loader)

- A simple CSS class for a loader animation
- HTML

```
<div class="loader"></div>
```
- CSS rule
  - Uses CSS variables
  - Rounded corners to create circle
  - CSS animation





## CSS Loaders & Spinners

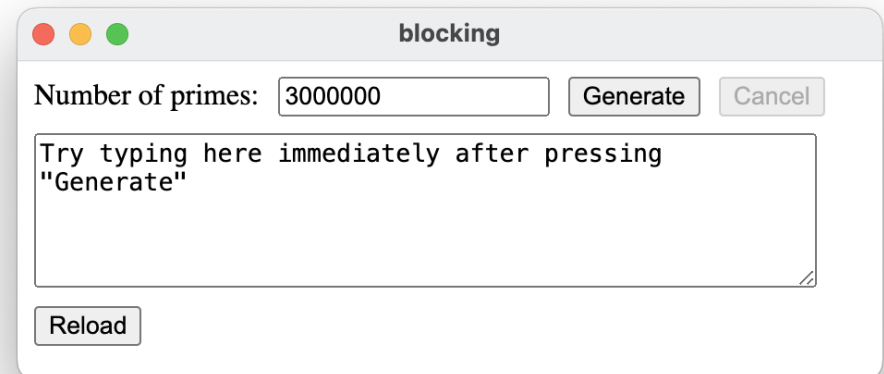
- <https://github.com/vineethtrv/css-loader>

# Handling (non-Web API) Long Tasks

- Goals
  - keep UI responsive
  - provide progress feedback
  - (ideally) allow long task to be paused or canceled

# blocking

- Shows what happens when long tasks NOT handled asynchronously
    - DO NOT DO THIS!
  - Demo
    - prime number generation code (intentionally inefficient)
    - dispatch blocked (try typing in textarea)
    - Cancel button unusable
    - Note even DOM update is blocked
- ```
output.textContent = "Starting ...";
```





# Threading

- Manage multiple concurrent threads with shared resources, but executing different instructions
- Threads are a way to divide computation, reduce blocking
- Concurrency risks: e.g. two threads update a variable
- Browsers support **worker threads**
  - dedicated workers
  - shared workers
  - service workers

# worker

- Uses web worker
  - create a dedicated worker
  - generate.ts has code for thread to execute
- `Worker.postMessage( ... )` to send message
- `Worker.addEventListener( ... )` to receive messages
- Messages from worker to main
  - main to thread: start  
[ "generate", 100000 ]
  - thread to main: progress  
[ "progress", 0.5 ]
  - thread to main: done  
[ "done", 100000 ]
- HTML progress bar

