Timers

Relevant Links

- Flanagan's book, Section 14.1
- The Event Loop[1]

Notes

The Event Loop

Javascript follows a simple concurency model, based on an "event loop":

- There is a *queue* of "messages".
- When an "event" occurs and a handler was registered for it, a message is added to the queue. These events could have come from a UI interaction, from a page loading, or programmatically via the timers we will discuss in a moment.
- As long as the queue is not empty, it proceeds to its next message and executes it, triggering all its function handlers in order.
- Execution of that message will continue until all the handlers have returned. There is no way for a function to pause its operation mid-way while some other function performs a task. This has consequences on page refreshes for example, as the page cannot refresh while a function is running.
- One important characteristic of this model is that it can never block. There is never a situation where one function is waiting for another to finish.
- An obvious disadvantage is that it does not really benefit much from multi-threading. However there is the concept of Web Workers[2] which we will hopefully get to later in the course.

Timers

There are a number of Javascript functions that allow us to add custom "events" on the event loop, to be executed at an appropriate time. Most of these take the form:

```
timerFunction(f, milliseconds);
```

Where we "register" the function `f` to be executed after a given number of milliseconds. This is a "desired" time: If something else is executing when that time comes, then the function will be queued to run at the next available opportunity.

You should not really expect the function `f` to receive any arguments, nor to have a good `this` object.

Here are the main methods:

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/EventLoop
[2] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

setTimeout[3] Registers a function to run once at some time in the future.
setInterval[4] Registers a function to run repeatedly every so often.
clearTimeout[5] **/ clearInterval**[6] These can be used to clear a previously set timer. They use the return values of the corresponding "set-" methods.

Here is a simple example:

```
var f = (function() {
  var i = 0;
  return function() {
    i += 1;
    console.log(i);
  }
}());
var id = setInterval(f, 1000);
// Watch the pretty numbers
// Run next line when you want it to stop
clearInterval(id);
```

Function Idioms

We look here at some function idioms that timers allow.

Delay This is a simple idiom. It is given a function `f` and a delay time in milliseconds, and returns a function that when called sets up a delayed call to `f`, passing along whatever arguments it is given.

This idiom can be handy if we want to pass a function to someone else to execute, but we want to make sure they can't run it right away.

```
function delay(f, milli) {
  return function delayed() {
    var context = this;
    var args = arguments;
    setTimeout(function() {
      f.apply(context, args);
    }, milli);
    return;
  };
}

// Sample run:
var g = delay(console.log, 5000);
g.call(console, "5 seconds later");
```

Throttle Given a function and an interval, it will only allow the function to be called once in that interval. Useful for events that are coming too fast for us to process them all.

```
function throttle(f, interval) {
  var available = true;
  // function "wait" used in setTimeout
  function wait() { available = true; };
  return function throttled() {
    if (available) {
      available = false;
      setTimeout(wait, interval);
      f.apply(this, arguments);
    }
  };
}

// Sample run:
var g = throttle(console.log, 5000);
g.call(console, "Prints right away"); g.call(console, "gets lost");
```

Debounce Given a function and an interval, it will only run the function after that amount of time has passed since the last time the function was invoked. So if you call the function it will start waiting for that amount of time, and if it is called again it will reset the timer and start waiting again.

Essentially this function will not run as long as it is getting called.

Useful when we have incoming input, and we only want the function to execute once that input is completed (say once someone has stopped moving their mouse or stopped typing).

```
function debounce(f, wait) {
  var timeout, context, args, lastCalled;
  function later() {
    var sinceLast = new Date() - lastCalled;
    if (sinceLast < wait) {
      timeout = setTimeout(later, wait - sinceLast);
    } else {
      timeout = null;
      f.apply(context, args);
      context = null;
      args = null;
    }
  }
  return function debounced() {
    context = this;
    args = arguments;
    lastCalled = new Date();
    if (!timeout) {
      timeout = setTimeout(later, wait);
    }
  };
}

// Sample run
document.onmousemove = function() { console.log("All the time!"); };
document.onmousemove = debounce(function() { console.log("only when I stop!"); }, 3000);
```