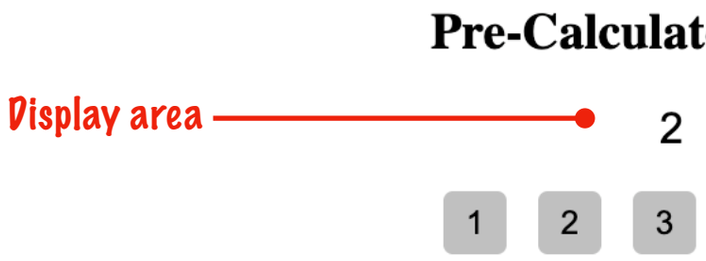
Exercise 18

Rev. 25Jan2019

BUILDING A "PRE-CALCULATOR"

Let's build a calculator! Just kidding. That's too hard. Instead, let's build a "pre-calculator" — sort of an early evolutionary branch on our way to (maybe) (eventually) building a full-blown calculator. This is what it will look like:



When a button is pressed, the value of that button will be shown in the display area. That's all we'll attempt with this exercise.

It's a good idea to examine the provided HTML files for all exercises. Studying them will give you a sense for how to think about structure.

Examine the HTML for **Exercise 18**. I draw your attention to the three `<div class="button">` elements. There's an additional property, `onclick`. We've used that before in a previous exercise. When we last saw it, the `onclick` property had a call to the `alert` function. That `alert` function is built into the browser itself (along with other functions like `querySelector` and `getElementById`).

This time, we're going to build our own function, which we'll name `display`. Inside the parentheses are the values for each individual button. Let's go to `index.js` and learn a bit about writing our own functions.

First thing: what is a function? It's a (typically) smallish bit of code that may accept some information sent into it — that's what we see in the HTML where we're passing in the value of the pressed button — and has some distinct function.

Let's start with a skeleton for a function:

```
let functionName = (arg1,...argN) => {
  // function "body" goes here
}
```

As we've done with other variables, we start with `let` followed by the variable name, followed by the `=` symbol. After that, things take a decided turn for the weird. But it looks much worse than it really is. Let's break down the parts of the skeleton.

THE ARGUMENTS

This is the pattern: `(arg1, ...argN)`. A function can take in any number of *arguments*, which are just values passed into the function. For each argument you expect, you provide an argument name. This will be much easier to understand with a couple of examples.

Arguments are one way we avoid hard-coding values, making them, instead, dynamic

Let's say we have a function named `greet`. The purpose of the function is to output "Welcome, *person*!", where *person* is replaced with the name of the person passed into the function. Here's how I'd write that:

```
let greet = (person) => {
  alert("Welcome, ' + person)
}
```

Note that the person's name being passed into the `greet` function is given an argument name of `person`. That allows me to use that variable inside the function body, which I do inside the call to `alert`.

Now, let's rewrite `greet` very slightly to accept a first name and a last name:

```
let greet = (firstName, lastName) => {
  alert("Welcome, ' + firstName + ' ' + lastName)
}
```

If you need 10 arguments, you would give each of them a distinct argument name.

CALLING FUNCTIONS

A function doesn't do anything until someone *calls* it. You already know this. In a previous exercise you *called* the `alert` function, passing into it an argument. The `alert` function existed before you called it, but

didn't "run" until you called it. It's the same with the `greet` function you've just seen. It sits waiting for a call to run.

There are several ways to call functions. Here's one way we might call the `greet` function:

Defining the "greet" function

Getting the person's name that we'll be passing into "greet"

Calling the "greet" function, passing into it the person's name

```
let greet = (person) => {
  alert("Welcome, ' + person)
}

let name = prompt("What is your name?")

greet(name)
```

Something that most people initially find confusing is the fact that the variable name that we got from calling the `prompt` function, and which is passed into `greet`, doesn't match the argument name of `person`. We do this because the function we write may be used throughout the application, and by other programmers who will have their own variable names that we can't know.

Imagine the mess trying to coordinate what the variable should be named that will be passed into the `greet` function! Instead, we take whatever *value* is passed in and *we* refer to it, within the function, by the name we give it when we declare what arguments are accepted. If this still seems hard to grasp, don't worry: it will sink in quickly once you begin working more with functions and, before long, *you'll* be trying to explain to someone why function argument names don't necessarily match the variable names passed into the function. I'm not altogether thrilled with the way I've explained it, so if you find a better way, let me know so that I can ~~steal it~~ give you credit for it.

So, we've seen that you can call a function directly, as we did in the last line of code above: `greet(name)`. Sometimes, though, we only want to call a function when some *event* happens. In our case, the event is when the user clicks the button. That's just what the `onclick` event handler does: when the user clicks a button, the `display` function is called, and is passed the value belonging to the clicked button.

```
<div class="button" onclick="display(1)">1</div>
```

- [ ] Now that you know all that, you can probably write the `display` function on your own. Try it! Don't worry: you won't break the internet if you fail, but you probably will learn something.

Feel free to google various odd things like "css em" or "css :hover". CSS is very forgiving and, though the elements might look strange, you won't break anything.

Of course, if you get stuck, refer to `answer.js`. But, really, try it on your own first because when you do look at my solution, you're going to say, "Dang! I could have done that." And you'll be right.

- Finally, take a look at the CSS for the HTML. There are some things to observe there. If you're like me, the more exposure you have to CSS, the better sense it will make. And the best way to *get* CSS is to play around with it. Change the values on various properties and see what that does to the styling of the elements.