I am actually not five years old. I am just starting to look seriously at ELM as a functional language. (Inspired by the "goodbye to FRP" article) There are so many ways people have worked with time and events functionally (I am sure that a list that includes tasks, futures, promises, signals, and now subscriptions is not exhaustive) So, who is willing to dive in and explain how subscriptions help me? I have been waiting for good documentation on this, but sometimes the crowd is a better teacher. So, from type signatures to syntax to semantics to insight into how to shape code, let's make this the best post in <a href=”<https://www.reddit.com/r/elm>” target=”\_blank”>[/r/elm](https://www.reddit.com/r/elm)</a>.

<b>ANSWER:</b>

Ok. So there's this thing called "communicating sequential processes" (CSP), first talked about by Tony Hoare. It's the idea of how to connect up communication within parts of a program - how to communicate between the pieces of your application in a nice, composable, good way that makes reuse possible and composability kinda fun. It's effectively core.async channels in clojure/script... if we model our app's communication in this way, it's very close to the original intention of objects (<http://c2.com/cgi/wiki?AlanKayOnMessaging>). Look into erlang's actor model, and the idea of having chunks of code "listening" to the things they care about. Perhaps OOP *should* have been called *Message-Oriented Programming*, but it wasn't and so we got stuck with confusion. We're only just now starting to come around to this realisation. <br/><br/>

Anyway, these are the basics. This (CSP) is "old magic" in terms of computer programming - leveraging papers from the past. Haskell has been doing this leveraging for a very long time (mostly from Math). Elm has had this kind of model, this CSP model that is very similar to Erlang's actor model, since the beginning with Signals and whatnot. These are of course now "gone" (at least from programmer visibility) in 0.17.0. What has replaced them? Well, nothing, because they're now baked in.

The closest thing to Elm that I've seen is an architecture explained within a project called re/frame on clojurescript, which uses reagent (which is a nice functional wrapper on react's rendering layer), and describes a way to communicate and talk about state. <https://github.com/Day8/re-frame> . <br/><br/>

The major difference between Elm and any of these other types of things is that Elm manages all the connection stuff for you. You don't have to spend the pain to learn about all this stuff, and "the right way" is already set up for you. And, if you've ever had to do one of these before, you know how much work this saves an experienced programmer. This stuff is usually boilerplate, and it *should* be in the language or framework, so it's very nice that it is with Elm. The current version of Elm very much has a *nice* well-thought-out architecture built into it that hides all the wiring of connecting the pieces together. This is the application architecture you *would want* if you did it yourself nicely. <br/><br/>

Facebook people (reactjs) have been doing similar things (after they saw Elm's stuff, from what I understand) as can be seen at <http://flowtype.org/docs/react.html> whereby there is a way to connect up the state-dependencies of your programs so messages (which can contain state) just flow along from top to bottom and back up to the top again, causing changes to the model and therefore the view as they do so. <br/><br/>

The beauty of this architecture is that you can compose these items, build software in a reusable way, and you have the inter-app communication already designed for you. This is exactly what you need if you want to build *big*applications, and it turns out that it's also very good for building small ones, too. <br/><br/>

When you write an Elm app, you write pure (in the functional sense; functions only depending on their arguments, and not some other piece of changing state, and not functions that can do anything other than return a value) code that explains declaratively what the view should look like (and not a set of functions on how to mutate it). The view responds *ONLY* to the model. <br/><br/>

You declare what kind of thing your model is, its intitial state, and then an "update" section: a function for adjusting the model, based on recognising (pattern-matching on) Message values. You hook these "adjustment functions" into your view by making your view have Messages, which are implicitly functions that call back into the update functionality. This is so that everything is described "as it is" rather than "as how to modify" - that is, without having to reason about sets of mutated state functionality in your application. Everything is clearly and cleanly described. So, for example, you might have a button in your view. Maybe you have a text field that shows an Int value, too. The Int comes from the model, but you don't have to put it there, you just place its variable in the view, the button has something like "send message increment" in its js-browser onClick handler. Now, that is actually just a message in Elm, but it translates into functionality in JS. You don't have to write the actioning functionality yourself, because Elm's onClick implicitly has that functionality that calls update in it: it takes your message value, and passes it into the update function when the JS onClick is called. The important thing is you don't have to think about this. You can program as if everything is "now" without worrying about crazy messy states things can get into. When you run this application, the update function will have code that simply increments the model's Int value when it is passed an Increment message, the view is hooked up in such a way to know what it depends on and when that changes, that it needs to update its view if anything has changed. Everything is declarative, so it's easier to think about, nothing gets out of sync. <br/><br/>

Subscriptions are one part of the managed effects of Elm. This is very similar to how Elm has the "managed events" of the update chain described above, only it's about *effects*. That is, how you interact with the very necessary kinds of values that change over time like "the wall-clock time" or "mouse movements" or "keyboard presses" or "random numbers", not static or single values, but ones that are more like a stream of values that change over time due to all kinds of external "stimuli". These kinds of values are what make our programs interesting and give them life. They're "inputs" in our Input/Processing/Output cycle. If your app subscribes to, say, mouse movements, you then hook this into your update function by a function that makes it create messages that will be described in the update function about what changes to the model should be enacted when mouse movements happen. That is, this is the "pouring water in the top" part of the flow of the architecture: it's how the inputs get turned into Messages. This is how non-pure values (ie values whose identity stays the same but whose value is different depending on something else) get pulled into our pure application's model, and therefore how they drive the view, and step the application forward. <br/><br/>

Now, how are they different from Signals? Well, there's a lot of overlap. Because all of the effects in Elm were previously very much *NOT* managed, you'd have to program their co-ordination yourself, which was three things: 1. Very hard to learn about for new people (Evan has said this is one of the biggest stumbling blocks). 2. Tricky to reason about (even for non-beginners). and 3. Actually "incidental complexity" and therefore unnecessary if you look at what the programmer is trying to achieve with them. <br/><br/>

Subscriptions grabs everything that was in Mailboxes and Signals, and takes it away from programmer control, and instead asks the programmer "What kinds of inputs, or feeds of information do you care about knowing about in your program?" and "How do you want me to create messages out of these things?" <br/><br/>

So, yes, Subscriptions are a way to inject messages into your app from "effects" that happen outside-your-app's context.

Take a look at this example: <http://elm-lang.org/examples/drag> ... and you'll see the subscriptions as:

<pre><b>

subscriptions : Model -> Sub Msg

subscriptions model =

case model.drag of

Nothing ->

Sub.none

Just \_ ->

Sub.batch [ Mouse.moves DragAt, Mouse.ups DragEnd ]

</b></pre>

If you look at the Just \_ part, you'll see the Sub.batch function which is being applied to two items,Move.moves DragAt creates DragAt messages with the particular co-ordinate data of the mouse, and Mouse.upscreates DragEnd messages.

Sub Msg values are implicitly hooked into the execution model such that they *generate* messages when the effect(s) that they depend on change.

Take a look at the update function to see how these play out into changes into the model. <br/><br/>

Then you can look at the view to see how these model states play out into the view.

Hope this helps and answers some of the questions you had about why Subscriptions are a better way to go.