List processing and functional Abstractions

Topics include

- Passing functions as arguments
- Declarative programming with abstractions
- List processing abstractions
- Iteration, filtering/selection, transformatio, interrogation, sort
- Combining abstractions
- Functional abstractions on objects
- Low level abstractions

Passing functions as arguments

Specialize abstractions

- Functions are first-class values, what does that mean? They can be stored and passed as arguments to other functions
- Example 0: Separate how an order is built, from the method that constructs the order for specific individuals. This is a separation of concerns that brings some abstraction out of code.

Functions as arguments

- When we iterate through an array, we can control which array to iterate through, but we CANNOT control 1) how to iterate through it, or 2) what to do with each element
- By abstracting the iteration process, we can re-use the same code over again in the same program, see Ex 1
- Method 1: we pass a `callback` argument into the function, for each element in the array, invoke callback and pass the element. This way, we can pass a function to the original `iterate` function to do the same thing. The `callback` function represents the argument that is later passed in through the method invocation, and specifies what needs to be done.
- Method 2: we can also save the specifics of what to do with the element in a function, and pass that into the `callback` argument. The difference is that the method invocation no longer has to specify the action required, it can just pass in the function that will specify exactly what needs to be done.

Behaviour as arguments

- What we know so far: passing objects into a function, where the function body creates the logic that works with the data to produce a result - either return value or side effects
- What is new now: the function can also take in a function expression, and this argument can provide more behaviour specification for the function. This can now not only provide the data to for example, iterate through, but can also "iterate through an array and do something"
- We can use an abstracted function to recreate the iteration functionalities, we can also use the built in method `#forEach` to do the same thing

Declarative programming with abstractions

What is the imperative style?

- Focused on the steps of solving a problem, what most programmers start off doing

Imperative stype with function abstractions

- Move certain details of implementation into another method, hides some abstractions and raises the abstraction level of program

Iteration focused abstraction with #forEach

- When the method responsible for iteration is also abstracted away

Filter abstraction that reflects purpose

- Now three levels of abstraction: 1. use built-in method `filter`, 2. log the function expression saved in a variable, 3. the `filter` method takes in an argument returned by a function that checks for odd numbers
- Ex3.js

Declarative programming

- Focuses on what to do, instead of how to do something. e.g CSS
- The higher the level of abstraction, the more declarative code looks like
- Try to build own abstractions to push down the implementation details, let code communicate intent

List processing abstractions

Here's a list of abstractions

- `Array.forEach`, returns `undefined`, iterates over an array, similar to `#each` which takes an array, iterates, and returns the input
- `Array.filter`, returns a new array, selects a subset of elements, similar to `#select` or `#reject`, which also returns a new array
- `Array.map`, returns a new array through transformation, similar to `#map`
- `Array.sort` returns the sorted array, and sorts in place, similar to `#sort` that does not mutate the input array
- `Array.every` and `Array.some` which interrogates array elements to see if they meet a condition
- These methods are named callback functions, since the methods "call back" the function

Iteration

- The functinon expression that expresses the "what needs to be done" to a selection can be saved to a variable function expression, or can be passed into the iteration directly as an anon function

Filtering/Selection

- Similar to `forEach`, `filter` takes a Function object, but only the argument that's called in the function is mandatory
- The function takes 3 arguments, current el, idx of current el, and array, and tests for each element, if true, included in return array
- There's also an increasing focus on surfacing abstractions
- In the filtering example, we have one filtering function that goes through a collection and filters out arrays that return `true` to specific conditions

- Then the main function can focus on the specific functionality requirements instead of repeating the iterating and selecting functions again

- 3 parts: 1. one part that does the filtering, 2. the specifics of the function (e.g. even or odd, multiples of 3), 3. the overall calling function that returns the specific functionality passed through the filtering function

Transformation

- The map function works similar to the each and select functions in the way it iterates over the input array
- Similar to the select function, a new array is created
- Note that it takes three arguments, function(array, index, array), but only one that's needed in the function itself is mandatory, the others are not
- Similar to the other iterators, this method can be used as a template generalize the mapping function, and can be used as a return function that takes in the input array and the function that specicifies the "what" of the transformation
- The function is implemented by requiring three parameters that specifies the input array, the function laying out what needs to be performed, and the initial value

Reducing

- Implementing the reduce method is slighly trickier because it takes in another argument, the accumulator, on top of the current element, current element index, and the array
- The function returns a type specified by the function that lays out the "what" to reduce
- The trick to implementing the program is to let another helper function specify exactly what needs to be reduced, and the actual reduce function is only concerned about iterating over the input and the initial value (if that's given)
- E.g. if we are stacking a tower, it takes in the bricks, points out the first brick to begin with, and provides a corner where the finished products should be placed it. It does NOT specify how the bricks should be stacked (horizontally, zig-zag, or vertically)

Interrogation

- The two common functions are `Array.prototype.some` and `Array.prototype.every`. Both invoke callback function for each element of the array iterated over, as other iterating functions do, the element, the index, and the original array.
- This just means that it has access to these three arguments for each element of the array.
- Check through the whole array for any element that does or does not satisfy a specific condition. The only difference is the exit condition when implementing the methods.
- The method, when used, will take two arguments, the array to iterate over, and the function that specifies the conditions for the function to check.
- The return value is either `true` or `false`

Sort

- The function takes one callback function, which specifies in which direction the sorting should happen. The callback function takes two arguments and compares them, returns -1, 0, and 1 and those determine the direction of the sort
- This mutates the array in place and can be used for side effect or its own return value

Combining abstractions

- The three-step example of finding the most frequently used initial in an array of names

- First get initials, then get hash of k-v pairs of initials and counts, then find the most frequently occuring initial

Functional abstractions on objects

- Objects key value pairs can be iterated over but not directly accessed like an array
- Thus, `Object.keys(ObjectHere)` is used to get an array of keys, and values can also be iterated over this