**Data Types & Composite Objects.**

1. Type Systems
   1. Type system dictates how we’re allowed to store information and how we can pass it around.
   2. Structuring our information via thoughtful patterns increases efficiency.
   3. Typing gives “meaning” to data – basically, how can JS handle this data? This depends on the type.
   4. Types
      1. Static v dynamic
         1. JS – dynamic;
            1. We can change the data type held by a variable.
            2. We can assign a var to a number, then later to a string.

Not as efficient at runtime, although nice for dev work.

* + - 1. **When** is data type information aquired?
      2. Static – the type of var is declared at initialization and cannot be changed.
         1. The compiler doesn’t need to make
    1. weak V strong
       1. JS – weak. What does this mean?
          1. A function that calls for “a + b” may return (4) when a = 3 and b =1
          2. But will return 1cat if a = 1 and b = “cat”

(*implicit* type coercion)

* + - * 1. In other words, JS is able to coerce types as needed.
      1. **How strictly** are the data types enforced?
      2. **Strong** types cannot be *implicitly* coerced.

1. JS’S type system
   1. Dynamic v weak
   2. Primitive v composites
      1. A primitive is **not an object** and **has no methods or properties.**
         1. But doesn’t string have methods and properties?
         2. “String object” – a special string object wraps itself around the string primitive, which gives it methods and properties.
         3. They are immutable
            1. When it is created, you cannot change its values,
            2. Although you can reassign the variable.
            3. Basically think “123” vs [1, 2, 3]
      2. Primitives are passed by **value,**  not by **reference**.
      3. 7 primitives:
         1. Number
         2. String
         3. Boolean
         4. Symbol
         5. Undefined
         6. Null
         7. BigInt
         8. A composite is any data type that can be constructed using the language’s primitive data types.
      4. Composites:
         1. Array
         2. Object
      5. These composite types **are mutable.**
      6. **Arrays** are great for ordered lists, but makes searching/adding/removing quite difficult.
         1. Think about trying to locate then second “1” in [1, 2, 3, 1, 4, 5, 1, 1]
      7. And these are passed by **reference**.
   3. Pass by Val vs Pass by Reference
      1. Primitive data types are **copied**
      2. Object data types are **referenced.**
   4. **Composite data types are stored in the “heap”**
      1. The “heap” stores the object, assigning **an address.**
      2. Within the global memory, we store the var name, but we assign to it not the object/array itself, but the **address where this can be found in the heap.**
2. Using Data Structures in JS
   1. New-ish JS features!
   2. ES6 (ECMAScript 2015)
   3. Set & Map
   4. Set
      1. This is a keyed collection
      2. That represents a unique set of values
      3. And has methods like add, has, delete, and foreach.
   5. Map
      1. Associates a value with a collection.
      2. With methods like get, set, has, and foreach.
      3. Keys can be any value (unlike a normal object)
3. Stacks & Queues
   1. For handling temporary data
   2. A stack is a list of els that are accessible from one end – the “top.”
      1. (push and pop)
      2. LIFO.
      3. But you CANNOT ACCESS RANDOM ELS in the stack.
      4. You might need this in something like a word document for an “undo” and “redo” button.
   3. Queues are similar to stacks; you cannot access random elements.
      1. Enqueue (add item to queue)
      2. Dequeue (take item off of queue)
      3. With array, adding is constant time (one step)
      4. Deleting is linear (gotta go through stuff, and time takes longer as it gets larger)
      5. With objects, enqueue is also constant time, and dequeue is linear.
      6. What if you had to program a printer??
         1. Well… you’d have a queue.
4. Linked Lists
   1. Linked lists are nested objects.
      1. We must keep track of the first node in order to access and traverse our list.
      2. We should keep track of the last node to give us a shortcut to add new nodes to the end.
      3. But the most useful thing is that adding/removing nodes is much much more efficient compared to arrays.
   2. Things like browser history, playlists, image carousels, all these things are stored as linked lists.
5. Hash Tables
   1. A hash table is used for data lookup at very large scales.
   2. You’ll hear them in system design and database management.
   3. This is a data structure (array) that stores info (data)
   4. Data is inputted at a specific location in the form of key-val pairs.
      1. The location is referred to as a “bucket”
   5. The hash function should convert some sort of immutable string into an integer.
      1. This integer becomes the array index where the keyval will be stored.
   6. Should map to unique numbers as much as possible with minimal collision.
      1. This is when two unique keys hash to the same value (into the same bucket)
   7. Consistently map a key to the same bucket. Every time you put in “dan,” you should get the same bucket number back.
6. Binary Search Trees
   1. Remember:
      1. Store nodes with lesser values on left, greater vals on right.
      2. Nodes which point to “null” are leaf nodes.
   2. This was the task in the binary search task in CSX.
   3. Depth first:
      1. Pre-order: check the node as you et to each node.
      2. Post-order: display on the last visit the node.
      3. In order: display in sorted by going down left side first.
   4. Breadth-first
      1. Traverse in level order where we visit each node on a level before going to a lower level.