DATA TYPES

Knowledge Has Organizing Power

Wholeness: All programs are organized in terms of data (information) and the operations that can be performed on it. *Science of Consciousness:* Programs are organized around different types of data and operations common to those types. Knowledge Has Organizing Power.

Main Points

1. Operations on primitive data types
    1. numbers
    2. strings
2. Array operations
    1. Modify
    2. Search
    3. Transform

Main Point Preview:

Numbers, strings, and booleans each have special operations such as parseInt, parseFloat, round, includes, and slice. Science of Consciousness: Knowledge is the wholeness of knower, known, and process of knowing. These operations represent processing on the data or the representation of some observation (known).

8 basic data types in JavaScript

PRIMITIVES

1. **Number** for numbers of any kind: integer or floating-point.
2. **String** for strings.
    1. A string may have one or more characters, there's no separate single-character type.
3. **Boolean** for true/false.
4. **null** for unknown values – a standalone type that has a single value null.
5. **undefined** for unassigned values – a standalone type that has a single value undefined.
6. **symbol** for unique identifiers.
7. **BigInt** for integers larger than 2**53

COMPLEX

1. **Object** for more complex data structures.
    1. Arrays and functions are objects

- The typeof operator allows us to see which type is stored in a variable.
    - Two forms: typeof x or typeof(x).
    - Returns a string with the name of the type, like "string".
    - For null returns "object" – this is an error in the language, it's not actually an object.

dynamic (loose) typing

- Dynamic typing

- JavaScript is a loosely typed or a dynamic language. Variables in JavaScript are not directly associated with a specific value type, and any variable can be assigned (and re-assigned) values of all types:

```
let   foo = 42;    // foo is now a number
foo     = 'bar'; // foo is now a string
foo     = true;  // foo is now a boolean
```

convert a string into an integer

- The parseInt() method converts a string into an integer (a whole number).
- two arguments.
  - string to convert.
  - radix. base for number system – e.g., 2 for binary, 10 for decimal

```
let text = '42px';
let integer = parseInt(text, 10);
// returns 42
```

- If parseInt encounters a character that is not a numeral in the specified radix
  - ignores it and all succeeding characters
  - returns the integer value parsed up to that point. parseInt
  - truncates numbers to integer values
  - If doesn't start with a number returns NaN

convert string argument to float

- parseFloat() method parses a string argument and returns a floating point

```
function circumference(r) {
  if (isNaN(parseFloat(r))) {
    return 0;
  }
  return parseFloat(r) * 2.0 * Math.PI;
}

console.log(circumference('4.567abcdefgh'));
// expected output: 28.695307297889173

console.log(circumference('abcdefgh'));
// expected output: 0
```

toFixed(n) rounds the number to n digits after the point

- Returns a string representation of the result.
- If the decimal part is shorter than required, zeroes are appended
- rounds up or down to the nearest value, similar to Math.round:

```
let num = 12.34;
console.log( num.toFixed(1) ); // "12.3"
```

```
let num = 12.36;
console.log( num.toFixed(1) ); // "12.4"
```

```
let num = 12.34;
console.log( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digits
```

Numeric conversion using a plus + or Number()

- Numeric conversion using a plus + or Number() is strict. If a value is not exactly a number, it fails:

- console.log(+"100px"); // NaN

+("1234") → 1234

Number("1234") → 1234

convert number to string

```
const foo = 45;
const bar = "" + foo;
const bar2 = "" + 108;
const bar3 = foo.toString();
const bar4 = 108..toString();  //need both periods after number
const bar5 = foo + "";
console.log(typeof foo === "number");  //true
console.log(typeof bar === "string");  //true
console.log(typeof bar2 === "string");  //true
console.log(typeof bar3 === "string");  //true
console.log(typeof bar4 === "string");  //true
console.log(typeof bar5 === "string");  //true
```

Rounding

- One of the most used operations when working with numbers is rounding.

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

strings are contained in quotes

- Strings can be enclosed within either single quotes, double quotes or backticks:
  - let single = 'single-quoted';
  - let double = "double-quoted";
  - let backticks = `backticks`;
- Single and double quotes are essentially the same.
  - Backticks allow us to embed any expression into the string, by wrapping it in ${…}:

```
function sum(a, b) {
  return a + b;
}
console.log(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

- Another advantage of using backticks is that they allow a string to span multiple lines:
- Our CS305 eslint convention is to require double quotes (backtick also ok)
  - Avoid’s confusion with apostrophes
  - Compatible with JSON

Searching for a substring

- look for the substr in str,
  - starting from the given position pos,
  - returns the position where the match was found or -1 if nothing can be found.

```
let str = 'Widget with id’;
console.log( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
console.log( str.indexOf('widget') ); // -1, not found, the search is case-sensitive
console.log( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

- includes, startsWith, endsWith
  - str.includes(substr, pos) returns true/false depending on whether str contains substr within.
  - right choice if we need to test for the match, but don’t need its position:

```
console.log( "Widget with id".includes("Widget") ); // true
console.log( "Hello".includes("Bye") ); // false
```

str.slice(start [, end])

- returns the part of the string from start to end (end not included)
  - If there is no second argument, then slice goes till the end of the string

```
let str = "stringify";
console.log( str.slice(0, 5) ); // 'strin', the substring from 0 to 5 (not including 5)
console.log( str.slice(0, 1) ); // 's', from 0 to 1, but not including 1, so only character at 0
```

- other helpful methods in strings:
  - str.trim() – removes (“trims”) spaces from the beginning and end of the string.
  - str.repeat(n) – repeats the string n times.

Main Point:

Numbers, strings, and booleans each have special operations such as parseInt, parseFloat, round, includes, and slice.

Main Point Preview:

Arrays are used in almost every program. There are special methods for common operations on them including to modify, search, and transform arrays.

- splice: modify an array
- slice: copy and return a piece of an array
- concat: combine arrays and elements into a new array
- forEach: execute a function using each element of an array

splice

- The arr.splice(str) method is a swiss army knife for arrays.
  - It can do everything: insert, remove and replace elements.

    arr.splice(index [, deleteCount, elem1, ..., elemN])

- It starts from the position index:
  - removes deleteCount elements and then
  - inserts elem1, ..., elemN at their place.
  - Returns the array of removed elements.

deletion:

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1); // from index 1 remove 1 element
console.log( arr ); // ["I", "JavaScript"]
```

splice (2)

remove 3 elements and replace them with the other two:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");
console.log( arr ) // now ["Let's", "dance", "right", "now"]
```

splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 2 first elements
let removed = arr.splice(0, 2);
console.log( removed ); // "I", "study" <-- array of removed elements
```

insert the elements without any removals.

```
let arr = ["I", "study", "JavaScript"];
// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");
console.log( arr ); // "I", "study", "complex", "language", "JavaScript"
```

slice

- returns a new array copying all items from index start to end
  - not including end

arr.slice(start, end)

```
let arr = ["t", "e", "s", "t"];
console.log( arr.slice(1, 3) ); // ["e", "s"] (copy from 1 to 3)
```

concat

- returns new array that includes values from other arrays and additional items
  - accepts any number of arguments – either arrays or values.
  - result is a new array containing items from arr, then arg1, arg2 etc.
  - If an argument argN is an array, then all its elements are copied.
  - Otherwise, the argument itself is copied. arr.concat(arg1, arg2...)

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4])); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6
```

Iterate: forEach

- run a function for every element of the array.
  - result of the function (if it returns any) is thrown away and ignored
  - Intended for some side effect on each element of the array
    - print or alert or post to database

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

- shows each element of the array

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(function(character){console.log(character)} );

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
   console.log (`${item} is at index ${index} in ${array}`);
});
```

Exercise

- use forEach to log all the even elements of an array to the console

[1,5,16,3, 108]

search an array

- indexOf/lastIndexOf and includes
- find and findIndex
- filter

indexOf/lastIndexOf and includes

- arr.indexOf, arr.lastIndexOf and arr.includes have same syntax and do essentially same as string counterparts
  - operate on items instead of characters:
  - arr.indexOf(item, from) – looks for item starting from index from, and returns the index where it was found, otherwise -1.
  - arr.lastIndexOf(item, from) – same, but looks for from right to left.
  - arr.includes(item, from) – looks for item starting from index from, returns true if found.

```
let arr = [1, 0, false];

console.log( arr.indexOf(0) ); // 1
console.log( arr.indexOf(false) ); // 2
console.log( arr.indexOf(null) ); // -1

console.log( arr.includes(1) ); // true
```

- uses === comparison.
  - So, if we look for false, it finds exactly false and not the zero.
- If we want to check for inclusion, and don't want to know the exact index, then arr.includes is preferred.

filter

- Apply function to each item in array and return new array of all that pass the filter

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
```

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

console.log(someUsers.length); // 2
```

find and findIndex

- find first element that satisfies a specific condition

```
arr.find(function(item, index, array)
  // if true is returned by passed function, item is returned and iteration is stopped
  // for falsy scenario returns undefined
```

- The function is called for elements of the array, one after another:
  - item is the element.
  - index is its index.
  - array is the array itself.

```
//Let's find the one with id === 1:
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
let user = users.find(item => item.id ===1);
console.log(user.name); // John
```

- arr.findIndex same but returns index where element found instead of element and -1 when nothing found.

Exercise

const numbers = [1, 5, 18, 2, 77, 108];

- use filter, find, and findIndex to find
  - all the even numbers
  - the first even number
  - the index of the first even number

transform an array

- map
- sort
- reverse
- reduce
- split / join

pure

destructive

split/join

*

- **split()** divides a String into an ordered list of substrings,
  - puts substrings into an array,
  - returns the array.
  - division done by searching for a pattern; provided as the first parameter

- **join()** creates and returns a new string by concatenating all the elements in an array
  - separated by commas or a specified separator string.

map

- one of the most useful and often used.
- calls function for each element and returns new array of results
- “map onto”
  - find or show connections between two things or groups of things
    - *map* brain functions *onto* brain structures
  - map the passed function onto each element of the array

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});
```

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // 5,7,6
```

Exercise

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // 5,7,6

//modify so that it logs array with index: item.length instead of just item.length
console.log("expect 0: 5, 1: 7, 2: 6", lengths);
```

sort(fn)

- default sort order is ascending and converts all arguments to strings
- sorts the array in place, changing its element order.
- returns sorted array, but the returned value is usually ignored, as arr itself is modified.

```
let arr = [ 2, 1, 15 ];
// the method reorders the content of arr
arr.sort();
console.log( arr );  // [1, 15, 2]
```

To use our own sorting order, we need to supply a (comparator) function as the argument of arr.sort().

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}
let arr = [ 2, 1, 15 ];
arr.sort(compareNumeric);
console.log(arr);   // [1, 2, 15]
```

Exercise

change comparator function to sort in descending order, then change it to sort in lexicographic descending order

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}
let arr = [ 1, 15, 2 ];
arr.sort(compareNumeric);
console.log(arr);
```

sort(fn) [2]

- comparison function is only required to return
  - positive number to say “greater” (i.e., a comes after b)
  - negative number to say “less”.
  - “greater goes to the right”
    - “increasing”
- That allows to write shorter functions:

```
let arr = [ 1, 2, 15 ];
arr.sort(function(a, b) { return a - b; });
alert(arr);  // 1, 2, 15
```

Remember arrow functions? We can use them here for neater sorting:

```
arr.sort( (a, b) => a - b );   //same as above
```

reduce

calculate a single value based on the array.

```
let value = arr.reduce(function(previousValue, item, index, array) {
  // ...
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

previousValue – is the result of the previous function call, equals initial the first time (if initial is provided).
item – is the current array item.
index – is its position.
array – is the array.

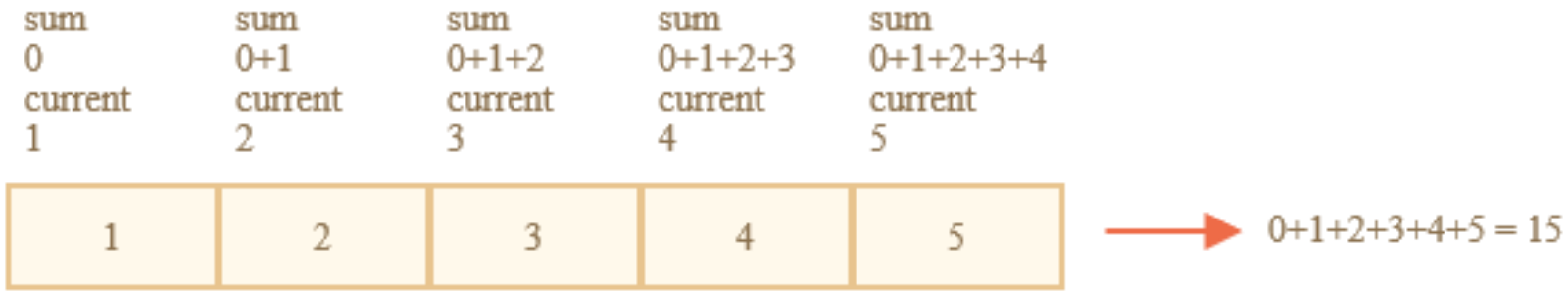
- first argument is the “accumulator” that stores the combined result of all previous execution.
  - at the end it becomes the result of reduce.
- CS305 convention: always include an initial value for clarity

reduce [2]

How to get a sum of an array in one line:

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce(function (sum, current) { return sum + current; }, 0);
let result2 = arr.reduce((sum, current) => sum + current, 0);
console.log(result); // 15
console.log(result2); // 15
```

- On the first run, sum is the initial value = 0, and current is first array element = 1
- On the second run, sum = 1, we add the second array element (2) to it and return.
- On the 3rd run, sum = 3 and we add one more element to it, and so on…



Exercise

- reduce the array to the product of the numbers (“expect 120”)
- reduce the array to the max of the numbers (“expect 5”)

```
let arr = [1, 2, 3, 4, 5];
```

Array methods summary

- To add/remove elements:
  - push(...items) – adds items to the end,
  - pop() – extracts an item from the end,
  - shift() – extracts an item from the beginning,
  - unshift(...items) – adds items to the beginning.
  - splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
  - slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  - concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.

- To search among elements:
  - indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.
  - includes(value) – returns true if the array has value, otherwise false.
  - find/filter(func) – filter elements through the function, return first/all values that make it return true.
  - findIndex is like find, but returns the index instead of a value.

- To iterate over elements:
  - forEach(func) – calls func for every element, does not return anything.

- To transform the array:
  - map(func) – creates a new array from results of calling func for every element.
  - sort(func) – sorts the array in-place, then returns it.
  - reverse() – reverses the array in-place, then returns it.
  - split/join – convert a string to array and back.
  - reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

map/filter/find/reduce are “pure” functions

- Important principle of “functional” programming
- Pure functions have no side effects
  - Do not change state information
  - Do not modify the input arguments
- Take arguments and return a new value
- Valuable benefits for automated program verification, parallel programming, reuse, and readable code

‘for in’ over object literal/Arrays –ES6

```
//for in over Object
//returns property keys (index) of object in
  each iteration - arbitrary order

var things = {
 'a': 97,
 'b': 98,
 'c': 99
};

for (const key in things) {
 console.log(key + ', ' + things[key]);
}
```

```
a, 97
b, 98
c, 99
```

‘for of’ vs ‘for in’ –ES6

- Both for..of and for..in statements iterate over arrays;
- for..in returns keys and works on objects as well as arrays
  - especially useful with “associative array” objects
  - easy to mistakenly use – check if get weird results
- for..of returns values of arrays but does not work with object properties

```
let letters = ['x', 'y', 'z'];

for (let i in letters) {
  console.log(i); } // "0", "1", "2",

for (let i of letters) {
  console.log(i); } // “x", “y", “z"
```

Summary ‘for’ loops

- ‘for’ is the basic for loop in JavaScript for looping
  - Almost exactly like Java for loop
  - Use this if you need the loop index
- ‘for in’ is useful for iterating through the properties of objects
  - especially useful for “associate array objects”
  - can also be used to go through the indices of an array (unusual use case)
- ‘for of’ is new convenience (ES6) method for ‘iterable’ collections
  - Array, Map, Set, String
  - Use this if usage involves a side effect and do not need loop index
- ‘forEach’ like for .. of but executes a provided function for each element.
  - forEach returns undefined rather than a new array
  - Intended use is for side effects, e.g., writing to output, etc.
- Best practice to use convenience methods when possible
  - Avoids bugs associated with indices at end points
  - map, filter, find, reduce best practice when appropriate

Main Point:

Arrays are used in almost every program. There are special methods for common operations on them including to modify, search, and transform arrays.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Knowledge Has Organizing Power

1. Numbers, strings, and arrays are important data types that have many common operations unique to their purpose and many methods in the language to support those operations.

2. JavaScript arrays are highly flexible data structures with many built in methods.

---

3. **Transcendental consciousness**. Is the experience of total knowledge and perfect orderliness.
4. **Impulses within the transcendental field:** Thoughts connected to the field of all the laws of nature will be supported by that level of total knowledge and coherence.
5. **Wholeness moving within itself:** In unity consciousness one appreciates daily perceptions and experiences as being infused with order and purpose.