# Prep Course

* Any unit of code that can be evaluated to a value is an **expression**.
* **Statements** are instructions to perform specific actions.

# Lesson 2: Small Programs

### Operations

* && and || are **short-circuit operators**.
* **Truthiness** – These values evaluate to false:
  + false;
  + undefined;
  + null;
  + 0;
  + ””;
  + NaN;

### Pseudocode

**START**: Start of the program.

**SET**: Set a variable.

**GET**: Retrieve input from user.

**PRINT**: Display output from a variable.

**READ**: Retrieve value from variable.

**IF/ELSE**: Show conditional branches in logic.

**WHILE**: Show looping logic.

**END**: End of the program.

### Flowcharts

|  |  |
| --- | --- |
| **START/STOP:** |  |
| **PROCESSING STEP:** |  |
| **INPUT/OUTPUT:** |  |
| **DECISION:** |  |
| **CONNECTOR:** |  |

## Coercion

* **Explicit Coercion:** Use specific methods to coerce values (i.e., Number(‘1’); )
  + Can coerce with unary operators + , - , ! .
* **Implicit Coercion:** Use == , != , or + between two variables.
  + When both operands are typeof “string” , JS compares lexicographically, if mixed it first coerces both to numbers.
  + When one operand is an object, both get converted to strings and combined.
* **Best Practices:**
  + Don’t use String() or .toString() in template literals.
  + Use unary + to convert strings to number.

### Coding Tips

* **Avoid magic numbers**: Pull numbers out into constants.
* **Regular**  while  **vs.**  while (true) **:**
  + Use while (true) **with internal break statement** if variable in while condition requires being defined before while statement.

## Variable Scope

* **Local Scope** (“Function Scope” or “Block Scope”):
  + Functions define new scope for local variables.
  + Nested functions define nested scopes.
* **Variable Shadowing (\*\*\*Avoid\*\*\*):** 
  + Inner and outer scope variables have the same name.
    - **Prevents access to outer scope variable.**
* Curly braces {} generally define new scope (e.g., if/else).

## Objects vs. Primitives

* Every value in JS is either a **primitive** or an **object**.
* Primitives:
  + “Atomic” values.
  + **Immutable**.
* Objects:
  + “Compound”.
  + Made up of primitives or other objects.
  + **Mutable**.

## Pass-by-Reference vs. Pass-By-Value

* Primitive values passed to function: **pass-by-value**.
* For objects, when an operation within a function mutates its argument, it affects the original object.
* Functions that mutate the caller are call **destructive** functions/methods.
  + Reassignment is not destructive.
  + Methods such as push() are destructive.

### Coding Tips

* Don’t use assignment in a conditional.
* Use \_ for unused callback parameters.

# Lesson 3: Practice Problems

* Empty array elements return undefined when called by index, but are actually empty:
  + (i.e., < n empty items> )
* **Deep Copy**: Makes a duplicate of every item in object.
* **Shallow Copy**: Only copies outermost (highest-level) values.
  + .slice() performs shallow copies.

# Lesson 4: JavaScript Collections

## Objects

* Two ways to reference element in object:
  + **Dot Notation:** obj.key;
  + **Bracket Notation:** obj[‘key’];
    - Need to use bracket notation if key is variable.
* Object keys cannot be duplicated; they get overwritten by the last one.

### Element Reference Gotchas

* Reference to an out-of-bounds array index returns undefined.
* Reference to an invalid object key returns undefined.
* Use .hasOwnProperty() or Object.keys().includes to distinguish between non-existent property and property set to undefined.
* Adding non-element properties does not increase array length.
  + Access these via object methods; in these cases element properties have index values converted to strings.
* typeof array = ‘object’
  + Use Array.isArray() to determine if variable is an array.
* String element reassignment is syntactically permitted but doesn’t affect the string.
* Use \ to escape character from syntax.
* str.charAt(n) returns ‘’ when out-of-range.
* str[n] returns undefined when out-of-range.

## Iteration

* **For/In Loops**: for objects
  + i.e., for (let currentPet in numberOfPets) {}
* Every loop can be rewritten as a generic while (true) loop.

# Lesson 4: JavaScript Collections

## PEDAC

### Problem / Examples:

Understanding the problem has three steps.

1. Read the problem description.
2. Check the test cases, if any.
3. If any part of the problem is unclear, ask the interviewer to clarify.

Questions to Ask:

1. Define terms *(e.g. ‘What is a palindrome?’)*
2. What to do with data types which appear to be unaffected?
3. How to deal with edge cases?

* Empty string, negative number, zero, etc.

1. Assumptions *(e.g. Can I assume all inputs are strings?)*
2. Other considerations *(e.g. Should I consider letter case when deciding if a word is a palindrome?)*
3. Do I need to return the same object or an entirely new one?

Inputs:

Outputs:

Describe rules to follow:

* Explicit requirements
* Implicit requirements

### Data Structure / Algorithm:

Should be able to write a plain English solution to the problem. If you can't do that, you won't be able to code it either.

## Selection & Transformation

* **Selection** results in *N* or fewer elements returned.
* **Transformation** always results in *N* elements returned.
* When performing a transformation, it's always important to pay attention to whether the original collection is mutated or if a new collection is returned.
* Even if we don't change any elements because none met our criterion it's still considered a transformation and called an **identity transformation**.
* *Note:* Trying to chain methods on empty collections or undefined is dangerous and results in a lot of broken programs.

## Array Methods

* Array methods take function as argument. Function expressions provided to other functions are **callbacks**.
* **Array destructuring assignment**: let [ key, value ] = keyValue;

| **Method** | **Action** | **Considers the return value of the callback?** | **Returns a new array from the method?** | **Length of the returned array** |
| --- | --- | --- | --- | --- |
| forEach | Iteration | No | No, returns undefined | N/A |
| filter | Selection/Filtering | Yes, its truthiness | Yes | Length of original or less |
| map | Transformation | Yes | Yes | Length of original |

## Arrays

### Elements & Returns

* ‘Elements’ added to array at invalid ‘indices’ are not elements (e.g. arr[-3] = 5;)
  + They do not increase the array length.
  + Can have array with length = 0 while still holding ‘elements’
    - use Object.keys(arr) to check contents.
* Number of elements in array isn’t necessarily the same as its length (can have gaps).
  + Elements shown in arrays as <n empty items> literally do not exist.
  + Empty items accessed by index return undefined though the value is not set.
* When a function doesn't explicitly return something, it implicitly returns undefined.

### Sorting

* Strings don't have access to any built-in sorting methods.
* When sort is called without arguments, it coerces all the array elements except those that are undefined to strings.
* undefined values are a special case when it comes to sorting. They are always placed at the end of the array no matter what the other values are.
* .sort() is destructive – mutates array and returns reference to existing array.
* .sort((a, b) => a – b);for ascending order.
* .sort((a, b) => b – a);for descending order.

### Nested Data Structures

* Shallow copy:
  + .slice();
  + Spread syntax: […arr];
* Deep copy:

let arr = [{ b: 'foo' }, ['bar']];

let serializedArr = JSON.stringify(arr);

let deepCopiedArr = JSON.parse(serializedArr);

* The JSON.stringify method **serializes** any object, including arrays, that only have primitives, arrays, and plain objects as elements. Serializing involves converting an object to a string form that can be subsequently converted back into an identical object.

# Lesson 5: Advanced Javascript Collections

## Sorting

* When sort() is called without arguments, it coerces all the array elements except those that are undefined to their string equivalents, then sorts them using string comparisons.
* Sort() is destructive. It doesn't return a new array; it sorts the original array, in-place, and returns a reference to that array.
* Can create a copy of the array using slice() and call the sort() method on the copy to avoid mutation.
* Sort() is an example of a function/method that has a side effect and returns a meaningful value at the same time. Do not emulate this in code.
* JS strings use UTF-16, not ASCII chars.
* Sort function: If the callback returns a number greater than 0, place b before a (i.e. return 1 indicates swap).

## Shallow Copy

* arr.slice();
* Spread syntax: […arr];

## Deep Copy

* JavaScript doesn't have an explicit method or function for deep copying objects.
* Can use:
  + let serializedArr = JSON.stringify(arr);
  + let deepCopiedArr = JSON.parse(serializedArr);

## First-Class Values & Higher Order Functions

* **First-class value** or **first-class object** is used to describe values that can be passed to functions as arguments or returned by those functions.
* In JavaScript, functions themselves are first-class values (can also pass them around your program like any other value).
* Sometimes refer to JavaScript functions as **first-class functions** to distinguish them from functions in other languages.
* For loops are **imperative** approach – explicitly describing each step to interpreter.
* Array methods are **declarative** approach – only possible because we treat functions as values.
* Functions that take other functions as arguments are called **Higher Order Functions**, as are functions that return other functions.
* Functions that we pass to other functions are often called **callbacks.**
* When evaluating code like this, ask the following questions:
  + What type of action is being performed? Method call? Callback? Conditional? Something else?
  + On what value is that action performed?
  + What is the side-effect of that action (e.g., output or destructive action)?
  + What is the return value of that action?
  + Is the return value used by whatever instigated the action?

# Array Methods

* Array.prototype.forEach( *callback* );
* Array.prototype.map(*callback*);
* Array.prototype.filter(*callback*);
  + Has selection criteria.
* Set to remove duplicates:
  + e.g., […new Set(arr)];
* Array.prototype.forEach( *callback* );
  + Returns **undefined**, does not do anything with returned value.
* Array.prototype.some( *callback* );
  + Returns **true** if any value is truthy.
* Array.prototype.every( *callback* );
  + Returns **true** if all values are truthy.
* Array.prototype.find( *callback* );
  + Returns **first truthy element** (or **undefined** if none are found).
* Array.prototype.findIndex( *callback* )
  + Returns first index or (or **undefined** if none are found).
* Array.prototype.indexOf( *value* )
  + Returns first index or (or **-1** if none are found).
* Array.prototype.lastIndexOf( *value* )
  + Returns last index or (or **-1** if none are found).
* Array.prototype.reverse()
  + **Mutates** the original array.
* Array.prototype.includes( *value* )
  + Returns **Boolean**.
* Array.prototype.fill( *value* )
  + Fills array section with value.
* Array.prototype.slice( *startIdx, ?endIdx* )
  + Returns new array with sub elements [startIdx, …, endIdx – 1].
  + Negative inputs treated as length + index.
* Array.prototype.concat( *value1, value2* )
  + Returns new array (**does not mutate**).

# Object Methods

* Object.assign( *obj1, obj2* )
  + Concatenates objects by adding key/value pairs to obj1.
  + Can copy objects with Object.assign({}, obj);
* Object.freeze( *obj* )
  + Returns **-1**.
* Object.isFrozen( *obj* )
  + Returns **Boolean**.
* Object.entries( *obj* )
  + Returns array with elements of [ key, value ].
* Object.hasOwnProperty( *key* )
  + Returns **Boolean**.

# String Methods

* String.prototype.startsWith()
  + Returns str[0]
* String.prototype.endsWith()
  + Returns str[ str.length - 1 ]
* String.prototype.charAt( *idx* )
  + Gets value at string index.
  + Returns empty string when idx out of range.
* str[ *idx* ]
  + Gets value at string index.
  + Returns empty string when idx out of range.
* String.prototype.slice( *startIdx, ?endIdx* )
  + Returns new string from str[startIdx] to str[endIdx – 1].
  + Negative inputs treated as length + index.
  + Infeasible calls return empty string.
* String.prototype.substring( *startIdx, ?endIdx* )
  + Returns new string from str[startIdx] to str[endIdx – 1].
  + Infeasible calls return empty string.
* String.prototype.includes( *value* )
  + Returns **Boolean**.
* String.prototype.match( *str | regex* )
  + */[regex]/g* to return array of all instances.
  + Non-global calls, looks for first instance.
    - Returns array: *[ str, index: idx, input: String, groups: undefined ]*
  + Returns **null** if nothing found.
* String.prototype.search( *str* )
  + Returns idx of first occurrence, else returns -1.
* String.prototype.padStart( *targetLength, padStr* )
* String.prototype.concat( *str1, str2* )
* String.prototype.trim( *str* )
  + Removes outer whitespace (“ “, \n, \t)
* String.prototype.trimStart()
* String.prototype.trimEnd()
* String.prototype.toUpperCase()
* String.prototype.toLowerCase()
* String.prototype.charCodeAt()
* String.fromCharCode()

# Miscellaneous Methods

* Number.isInteger( *n* )
  + Returns **Boolean**.
* Number.isNaN( *n* )
  + Returns **Boolean**.
* let regex = new RegExp(word, 'g');