Collections and Control Flow

CIS 1962 (Fall 2025)
September 15th, 2025

Lesson Plan

Weekly Logistics

Homework 1: Data Analysis RELEASED

- https://upenn-cis-1962.vercel.app/1-data-analysis
- Due on September 29th @ 11:59 PM on Gradescope!
- Please use Ed Discussion to ask homework questions or attend office hours (posted on the class website).
- Please don't share code publicly on Ed Discussion! Set your post private if you need to share code.

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

Collections

What are Collections?

Collections are data structures that allow you to group values together, allowing for more data organization and easier iteration.

There are 4 primary collections in JavaScript:

- Arrays
- Objects
- Maps
- Sets

Arrays

JavaScript **Arrays** is a 0-indexed collection of values that is:

Ordered

Arrays have a defined order that can be accessed using an index starting at 0 and ending at length - 1.

```
let arr = [1, 2, 3]
console.log(arr[0]) // 1
```

Dynamically-Sized

Arrays can grow or shrink in size by certain built-in methods.

```
let arr = [1, 2, 3]
arr.push(4) // 1, 2, 3, 4
arr.pop() // 1, 2, 3
arr.pop() // 1, 2
```

Mixed Type

Arrays can contain a mixture of types, as they do not have a defined singular type mandated of all elements.

```
let arr = [1, “2”, true,
{name:”v”,
role:”teacher”}]
```

Arrays and JavaScript Built-In Methods

JavaScript has built-in methods, functions that are properties of objects, for certain items, including all the collections.

Methods are called using dot notation (object.method()), and can take arguments.

For example, some array methods include:

- `.push(value), .pop(), .filter(callback), join()`

Array Methods: Access

Arrays are 0-indexed, and can be accessed using [].

Accessing items outside the existing indices of the array will return `undefined`.

```
let arr = [“a”, “b” , “c”, d”]

console.log(arr[0]) // a
console.log(arr[1]) // b
console.log(arr[2]) // c
console.log(arr[3]) // d
console.log(arr[4]) // undefined
console.log(arr[-1]) // undefined
```

Array Methods: Length and Setting

You can set the item at a certain index of an array using the index, like so:

```
arr[index] = desiredItem
// For instance:
let arr = [1, 2, 3]
arr[5] = 20
console.log(arr) // [ 1, 2, 3, <2 empty items>, 20 ]
```

This can increase the length of the array, as seen above

You can verify the length of the array using `array.length`:

```
let arr = [1, 2, 3]
console.log(arr.length) // 3
```

Array Methods: Insertion/Deletion

`push(value)`: insert an element **or** elements to the **end**, returns new length

`unshift(value)`: insert an element **or** elements to the **front**, returns new length

`pop()`: removes **and** returns the last element

`shift()`: removes **and** returns the first element

```
let arr = [“a”, “b” , “c”, d”]

console.log(arr.push(“e”)) // arr = [a, b, c, d, e], returns 5
console.log(arr.push(“f”, “g”)) // arr = [a, b, c, d, e, f, g], returns 7
console.log(arr.pop()) // arr = [a, b, c, d, e, f], returns g
console.log(arr.shift()) // arr = [b, c, d, e, f], returns a
console.log(arr.unshift(“a”)) // arr = [a, b, c, d, e, f], returns 6
```

Array Methods: Finding Elements

`indexOf(value)`: Find the index of a certain value, -1 if it doesn't exist

`includes(value)`: Check if and array contains a certain value

```
let arr = [1, 2, 3]

console.log(arr.indexOf(2)) // 1
console.log(arr.indexOf(4)) // -1
console.log(arr.includes(4)) // false
```

Array Methods: Slicing and Splicing

`slice(start, end)`: Returns a **new** array from some starting index to some ending index. If no ending index is specified, slices from the starting index to the last index. **Does not modify the original array.**

`splice(start, deleteCount, item(s))`: **Modifies the original array** by adding, removing, and replacing elements in place. **Returns removed elements.**

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

let slicedArr = arr.slice(2)
console.log(slicedArr) // [3, 4, 5]

arr.splice(2, 0, 6) // at index 2, insert item 6
console.log(arr) // [1, 2, 6, 3, 4, 5]

arr.splice(3, 3, 7, 8, 9) // at index 3, replace
the next 3 elements with 7, 8, and 9
console.log(arr) // [1, 2, 6, 7, 8, 9]
```

Arrays and Shallow Copies

Any array copying and methods that use copies of arrays create only **shallow copies.**

This means that only the **top-level references** are copied to the array, **nested references** to objects will still reference the original

```
let original = [1, 2, { a: 5 }];
let copy = original.slice();

copy[0] = 99;  // Only affects the copy
copy[2].a = 42; // Affects both

console.log(original) // [1, 2, { a: 42 }]
console.log(copy) // [99, 2, { a: 42 }]
```

Penn Engineering

Array Methods: Callbacks

There are certain array methods that take in **callback functions** as arguments.
We will discuss callback functions and higher order functions later.

These methods are often used with functions with arrow notation to define their behavior concisely.

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

// map
let mapped = arr.map((x) => x * x)
console.log(mapped) // [1, 4, 9, 16]

// filter
let even = arr.filter((x) => x % 2 == 0)
console.log(even) // [2, 4]
```

POLL QUESTION

```
let arr = [1, 2, 3, 4, 5];
let sliced = arr.slice(1, 4);
let spliced = arr.splice(1, 2,
'a', 'b');
```

What are the final values of arr, sliced, and spliced after execution?

arr: [1, 'a', 'b', 4, 5],

sliced: [2, 3, 4],

spliced: [2, 3]

Objects

Objects are data structures that store data in a collection of **key-value pairs**, or **properties**. They are similar to dictionaries in Python or HashMaps in Java.

Objects are the basis of **JSON** (JavaScript Object Notation), used widely in web APIs and data storage.

Objects are typically **unordered**, though many engines will preserve insertion order for keys.

Objects: Keys and Values

KEY :

- **Data Type**: Must be a **string** or **symbol**, all other types are coerced into strings

VALUE

- **Data Type: Any,** can store primitives and other objects and arrays

Creating Objects

There are 3 typical ways of making objects:

Object Literal

```
let animal = {
      name: “red panda”,
      formal: “ailurus
fulgens”,
      endangered: true,
      population: 15000
};
```

new Object()

```
let animal =
      new Object();
animal.name =
      “red panda”,
animal.formal =
      “ailurus fulgens”,
animal.endangered = true,
animal.population = 15000
```

Constructors (ES6)

```
class Animal {
        constructor(name, formal,
            endangered, population)  {
                this.name = name
                this.formal = formal
                this.endangered =
            endangered
                this.population =
            population
        }
}
let redPanda = new Animal(“red panda”,
“ailurus fulgens”, true, 15000)
```

Object Properties

Properties of an object can accessed, added, and deleted using either **dot** notation or **bracket** notation.

```
let account = {}
account.username = “Klein”
account.password = “*********”

console.log(“Resetting password
for ” + account.username)

delete account.password
```

Dot Notation

```
let account = {}
account[“username”] = “Klein”
account[“password”] = “*********”

let key = “username”

console.log(account[key]) // Klein
```

Bracket Notation

Object Properties: Methods

`Object.keys(object)`: list out the keys of an object
`Object.values(object)`: list out the values of an object
`Object.entries(object)`: list out the entries of key/value pairs of an object

```
let account = {}
account.username = "Klein"
account.password = "*********"

console.log(Object.keys(account)) // [‘username’, ‘password’]
console.log(Object.values(account)) // [ 'Klein', '*********' ]
console.log(Object.entries(account))
// [ [ 'username', 'Klein' ], [ 'password', '*********' ] ]
```

Functions in Object Properties

Because functions are first class objects, they can be stored as properties within objects.

```
let calculator = {
     add: function(x, y) { return x + y; },
     subtract: function(x, y) { return x + y; },
     multiply(x, y) { return x * y },
     divide(x, y) { return x / y },
     power: (x, y) => { return x ** y },
}

console.log(calculator.add(2, 3)) // 5
console.log(calculator.multiply(2, 3)) // 6
console.log(calculator.power(2, 3)) // 8

calculator.mod = function(x, y) {
     return x % y;
}
console.log(calculator.mod(10, 3)) // 1
```

Penn Engineering

Objects and JSON

JSON, an often-used format for data storage, looks like a JavaScript object literal, but is **entirely a string.**

Any keys are always enclosed in double quotes (“”):

```
let user = {
      name: “Voravich”
}
```

JavaScript

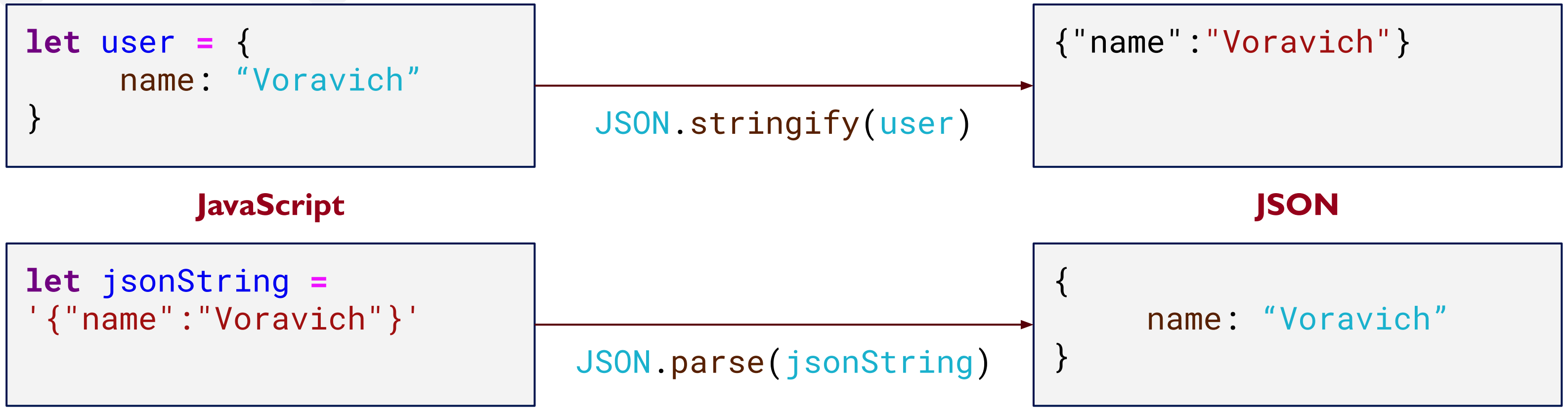
```
{
      “name”: “Voravich”
}
```

JSON

JSON: Serialization and Parsing

The process of transferring from object to JSON is called **serialization**, and can be achieved with `JSON.stringify()`.

The opposite process, transferring JSON to a JS object, is called **parsing or unserialization**, done through `JSON.parse()`.



```
let user = {
      name: “Voravich”
}
```

JSON.stringify(user)

```
{"name":"Voravich"}
```

JavaScript

JSON

```
let jsonString =
'{"name":"Voravich"}'
```

JSON.parse(jsonString)

```
{
      name: “Voravich”
}
```

JSON

JavaScript Object

POLL QUESTION

```
let fruit1 = { name: "apple" };
let fruit2 = { name: "apple" };
let basket = {};

basket[fruit1] = 10;
basket[fruit2] = 20;

console.log(basket[fruit1]);
```

What will be logged to the console?

20

Maps

Maps are an ES6 collection that, similar to objects, stores key-value pairs.

The difference from objects is that **map keys can be any type**, and maps are guaranteed to **maintain insertion order**.

```
let map1 = new Map();

let states = new Map([
     [17, “OH”],
     [6, “MA”],
     [11, “NY”],
])
```

Map Methods and Example

`set(key, value)`
Add or update entry

`get(key)`
Retrieve value by key

`has(key)`
Returns **`true`** if key exists

`delete(key)`
Remove entry by key

`clear()`
Remove all entries

`size`
Property — current number of entries

```
let productA = {
     id: 1,
     name: "Laptop"
};
let productB = {
     id: 2,
     name: "Mouse"
};

let shop = new Map();

shop.set(productA, 15);
shop.set(productB, 40);

console.log("Shop has: " +
shop.get(productA) + " laptops.")
console.log("Does shop have mice? "
+ shop.has(productB));
console.log("Total unique items:" +
shop.size);
```

Sets

Sets are an ES6 collection that are most often used to store **unique values**. Sets maintain insertion order.

`add(value)`: Add a value

`has(value)`: Returns `true` if value exists

`delete(value)`: Remove a value

`clear()`: Remove all values

`size`: Property — current number of values

```
let fruitBasket = [“apple”, “banana”,
“banana”, “orange”,”peach”, “apple”];

let uniqueFruits = new Set(fruitBasket);

uniqueFruits.add(“cherry”)
uniqueFruits.add(“banana”) // ignored

console.log(uniqueFruits.has(“cherry”))
// true
console.log(uniqueFruits.size) // 5
```

Destructuring

Destructuring is concise syntax for extracting values from arrays, strings, properties from objects.

Think of it as "unpacking" an array, string, or object, allowing you to extract only what you need from an object.

Destructuring is **non-destructive**, meaning the original object is not affected. The syntax is position-based, as seen below:

```
const arr = [10, 20, 30];
const [a, b, c] = arr;
console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

Destructuring: Arrays

```
const arr = [10, 20, 30];
const [a, b, c] = arr;
console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

Basic Use

```
const arr = [10, 20, 30];
const [a, c] = arr;
console.log(a); // 10
console.log(c); // 20
```

Get only first two elements

```
const arr = [10, 20, 30];
const [a, , c] = arr;
console.log(a); // 10
console.log(c); // 30
```

Skipping Elements

```
const arr = [10];
const [a, b = 40] = arr;
console.log(a); // 10
console.log(b); // 40
```

Default Values

```
const arr = [10, 20, 30];
const {[2]: a, [0]: b,
[1]: c} = arr;
console.log(a); // 30
console.log(b); // 10
console.log(c); // 20
```

Computed Property Names

```
const arr = [10, 20, 30];
const [a, ...b] = arr
console.log(a); // 10
console.log(b); // [20, 30]
```

Rest Syntax

Destructuring: Objects

```
const user = { name: "Adam", age: 51 };
const { age, name } = user;
console.log(name); // "Adam"
console.log(age); // 51
```

Basic Use

```
const user = { name: "Adam", age: 51 };
const { name: firstName } = user;
console.log(firstName ); // "Adam"
```

Renaming Properties/Alias

```
const user = { name: "Adam", age: 51 };
const { name, age, country = "US"} = user;
console.log(name); // "Adam"
console.log(age); // 51
console.log(country); // US
console.log(user); // { name: 'Adam', age:
51 }
```

Default Properties
Note that the original user doesn't change!

```
const user = { name: "Adam", age: 51,
country: "US"};
const { name, ...others} = user;
console.log(name); // "Adam"
console.log(others); // { age: 51, country:
'US' }
```

Rest Syntax

Penn Engineering

Destructuring: Function Arguments

```
let arr = [2, 3, 5, 7, 11];

function sumFirstTwo([a, b]) {
  return a + b;
}
console.log(sumFirstTwo(arr)); // 5
```

Arrays

```
let user = {
    name: "Bob",
    pronouns: "he/him",
    likes: "baseball"
}
function introduce({ name, pronouns,
likes}) {
  return `My friend’s name is ${name},
${pronouns.split("/")[0]} loves
${likes}!`;
}

console.log(introduce(user)); // "My
friend's name is Bob, he loves baseball!
```

Objects

Penn Engineering

Spread Syntax

The **Spread** syntax, denoted by **…** followed by any **iterable** (array, string, or object) can be used where zero or more arguments or elements are required, such as an array index or the inside of an object.

It's useful for:

```
let arr1 = [1, 3, 5]
let arr2 = [2, 4, ...arr1, 6]
console.log(arr2) // [2, 4, 1,
3, 5, 6]
```

expanding

combining

```
let arr1 = [1, 3, 5]
let arr2 = [2, 4, 6]
console.log([...arr1, ...arr2]) //
[1, 3, 5, 2, 4, 6]
```

```
let arr = [1, 3, 5]
let copy = [...arr]
console.log(copy) // [1, 3, 5]
```

copying

Rest Syntax

The **Rest** syntax, denoted by **...** followed by a **variable** name collects the remaining elements or properties into a new array or object, denoted by the given variable name.

Rest must go **last** wherever it is used.

It often used for:

```
function doSomething(a, ...b) {
     console.log(a) // 3
     console.log(b) // [1, 0, 1]
}

doSomething(3, 1, 0, 1)
```

Gathering variable number of arguments

```
let [first, ...others] =
                     [2, 3, 5, 7, 11]
console.log(first) // 2
console.log(others) // [3, 5, 7, 11]
```

Gathering remaining array elements

POLL QUESTION

```
const config = {
    title: "Dashboard",
    options: {
        width: 300,
        height: 200
    },
    theme: "dark"
};
```

How would you extract the title, width, and theme?

```
const { title, options: { width }, theme: colorScheme } = config;
```

POLL QUESTION

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };

const merged = { ...obj1, ...obj2,
b: 5 };
```

What is the value of `merged`?

```
{ a: 1, b: 5, c: 4 }
```

POLL QUESTION

```
const user = {
  id: 42,
  name: "Kai",
  email: "kai@email.com",
  role: "admin",
  meta: {
    active: true
  }
};
const { id, name, ...details } = user;
const { email, ...rest } = details;
```

What is the final value of rest?

```
{ role: "admin", meta: { active: true } }
```

5-Minute Break!

Penn Engineering

Control Flow

Penn Engineering

What is Control Flow?

Control Flow is the order of statements and function calls that a program executes.

Control flow allows us to make dynamic decisions and repeat actions depending on data and user input.

In JavaScript, control flow comes 2 basic forms:

- Conditional Statements (if… else, switch, ternary)
- Iteration (Loops)

Conditional Statements

Sometimes, we want our code to perform differently depending on certain conditions or data.

For instance:

- If it is a leap year (`year % 4 === 0`), add February 29th to the calendar
- If a user has input a username and password, allow a login button to be clicked
- Depending on a username's first letter, display a different tab on the webpage.

Types of Conditional Logic

```
if (temp > 35) {
      console.log("It's hot")
} else if (temp < 5) {
      console.log("It's cold")
} else {
      console.log("It's just 
right")
}
```

if, else if, else

```
switch (new Date().getDay()) {
      case 0:
            console.log("Sunday");
            break;
      case 1:
            console.log("Monday");
            break;
      ...
}
```

switch

```
let greetingMessage = isLoggedIn
      ? `Welcome back, ${user.username}.`
      : `Welcome, please log in.`;
```

ternary

Penn Engineering

If-Else Statements

If-Else Statements allow execution of different code blocks depending on whether certain conditions evaluate to be true.

- "If" is used to start a statement
- "Else If" allows additional, mutually exclusive conditions
- "Else" allows for a condition that executes if all others are false

```
if (condition) {
     // do something
} else if (anotherCondition) {
     // do something else
} else {
     // if all other conditions are false, do something
}
```

Evaluating Conditions

Conditions inside if-else statements must produce **booleans**, either **true (truthy)** or **false (falsy).**

JavaScript coerces values to be boolean if they aren't already.

```
if (number > 5)
```

```
if (username) // does username exist?
```

```
if (isButtonActive)
```

```
if (!userInput) // is there no user input?
```

```
if (username === "Eunsoo")
```

```
if (array.length) // are there elements in
the array?
```

✗

```
if (data !== null)
```

✓

```
if (!data)
```

Nested If-Else Statements

If-Else Statements can be nested for more complex logic.

Be wary of readability when using nested if-else statements!

```
if (user) {
  if (user.isActive) {
    if (user.role === “student“) {
      console.log(“Showing student view“);
    } else if (user.role === “professor“) {
      console.log(“Showing professor view“);
    } else {
      console.log(“Showing guest view");
    }
  } else {
    console.log("Account not active.");
  }
} else {
  console.log("Please log in.");
}
```

Switch Statements

Switch Statements allow conditional logic based on the evaluation of a **single variable or expression**, split into cases.

The statement try to match one of the defined cases, otherwise the defined default case will be run.

```
switch (expression) {
      case value1:
            // do something
            break;
      case value2:
            // do something else
            break;
      ...
      default:
            // fallback logic
}
```

Penn Engineering

Switch Statements: Break

Switch Statements work by evaluating the expression and comparing it to each defined case.

Because every case is checked, we will need to use `break` or else it will "fall through" to other cases!

```
let num = 7
switch (num) {
      case 6:
            console.log("num is 6")
      case 7:
            console.log("num is 7") // prints
      case 8:
            console.log("num is 8") // also prints!
}

// Both case 7 and case 8 are printed because break
is omitted! Don't forget to use break!
```

Switch Statements: Grouping

You can intentionally let cases fall through to **group** the cases together for easier readability.

```
let letter = “e”
switch (letter) {
     case “a”:
     case “e”:
     case “i”:
     case “o”:
     case “u”:
          console.log(“vowel”);
          break;
     default:
          console.log(“consonant”);
          break;
}
```

Ternary Operators

The **Ternary Operator** is a 3-operand operator that allows for concise conditional logic.

```
condition ? valueIfTrue : valueIfFalse
// if condition is true, then
valueIfTrue, otherwise valueIfFalse
```

This allows conditional logic to be inserted inline into complex expressions or return items conditionally.

Ternary Examples

```
// if/else
let canDrink = “”;
if (age >= 21) {
     canDrink = “Yes”;
} else {
     canDrink = “No”;
}
// Ternary
let canDrink = (age >= 21) ? “Yes” : “No”
```

Simplifying if-else logic

```
let grade = (grade >= 90) ? "A" :
               (grade >= 80) ? "B" :
               (grade >= 70) ? "C" : "F";
let gradeDisplay = `<p> Your Grade is: ${grade}<p>`
```

Nesting conditions, inserting dynamic content into HTML

POLL QUESTION

You're designing a function that takes a user's subscription type as a string ("free", "basic", "pro", or "enterprise") and returns a different welcome message for each.

What control flow structure would be best to use?

```
Switch statement
```

POLL QUESTION

```
const userStatus = "pending";
const result = userStatus === "active"
    ? "User is active"
    : userStatus === "inactive"
        ? "User needs activation"
        : userStatus === "banned"
            ? "User is banned"
            : userStatus === "deleted"
                ? "User not found"
                : "Unknown status";

console.log(result);
```

What is the output of this program?

“Unknown status”

What are Loops?

Sometimes, we want repetitive actions to be performed within code. For instance, we may want to:

- Perform an action on every item in a collection
- Find the maximum value within a collection
- Validate that all elements within a collection
- Constantly check for updates in a game
- Display all the keys or properties of an object

Loops are control flow structures that allow repetitive execution of lines of code as long as a certain defined condition holds true.

For Loops

For loops iterate a set amount of times, and are useful for tasks where indexes are important or you have a finite amount of times you want to execute code.

```
for (initialization; condition; update) {
     // loop code
}
```

For loops have 3 main parts:

- **Initialization**: executed once, sets a variable before the loops starts
- **Loop Condition**: defines the condition for this loop to run, will run the loop if true, and leave the loop if false
- **Update**: executed every time after code block finishes, often changing the value of the variable set in initialization

For Loop Example

```
let students = [
     {name: “Tyson”, grade: 89},
     {name: “Sam”, grade: 78},
     {name: “Chen”, grade: 96}
]

let sum = 0;

for (i = 0; i < students.length; i++) {
     sum += students[i].grade;
     console.log(students[i].name + “ has
     grade: “ + students[i].grade)
}

let average = sum / students.length;

console.log(“Average grade: “ + average);
```

For Of Loop

For of loops allow you to define a readable loop over any **iterable** object.

What is iterable?

- Arrays
- Strings
- Maps
- Sets

```
for (variable of iterable) {
      // loop code
}
```

```
let fruits = ["apple", "banana", "cherry"];
for (const fruit of fruits) {
      console.log(fruit);
      // apple, banana, cherry
}

for (const char of "hi") {
      console.log(char); // h, i
}

Let roomMap = new Map()
roomMap.set(101, "alice");
roomMap.set(102, "bob");
roomMap.set(103, "carol");

for (const [room_id, name] of roomMap) {
      console.log(`Room: ${room_id}, Name:
${name}`);
}
```

For In Loop

For In Loops allow you to define iteration over **enumerable** items.

What is enumerable?

- Keys of Property names
- Indexes of Arrays

```
for (key in object) {
      // loop code
}
```

```
for (index in array) {
      // loop code
}
```

```
let person = { name: "Ada", age: 31 };
for (const key in person) {
      console.log(key, person[key]);
      // name Ada
      // age 31
}

let arr = ["a", "b"];
for (const idx in arr) {
      console.log(idx, arr[idx]);
      // 0 a
      // 1 b
}
```

While Loops

While loops iterate until the loop condition is false, useful for tasks where you are unsure when the loop will stop executing, or for persistent tasks.

```
while (condition) {
     // loop code
}
```

```
let input = "";
while (input !== "yes") {
     input = prompt("Type yes:");
}
```

```
while (true) {
     console.log(“To infinity and 
beyond!!”)
} // Be careful of infinite loops!
```

Do While Loop

Do While Loops will execute the loop body at least once. This is useful if it is possible to never run a while loop, but you need the loop body to at least execute once.

```
let n = 0;
while (n > 0) {
     console.log("while:", n);
     n--;
}
```

Loop Never executes

```
let n = 0;
do {
     console.log("do-while:", n);
     n--;
} while (n > 0);
```

Executes properly (once)

Loop Control: Break and Continue

The `Break` keyword can be used to leave a loop early.

The `Continue` keyword can be used to skip the rest of the code in the iteration to move onto the next iteration.

```
// Find first number > 50
const nums = [0, -1, 120, -5, 20, 60, 8];

for (let n of nums) {
      if (n < 50) continue;
      if (n > 50) {
            console.log(n);
            break;
      }
}
```

POLL QUESTION

```
const sets = [
  [1, 2],
  [0, 1],
  [3],
  [4, 5, 6],
];

let sum = 0;
for (const [a, b = 1] of sets) {
  sum += a + b;
}

console.log(sum);
```

What will be logged to the console by the variable sum?

17

POLL QUESTION

```
let items = [{ count: 2 }, { count: 1 }, {
count: 0 }];
let idx = 0;
let operations = 0;

while (items[idx] && items[idx].count--) {
  operations++;
  idx++;
}

console.log(operations);
```

What is the final value of operations printed out?

2

Higher Order Functions

What are Higher Order Functions?

Higher Order functions are functions take other functions as arguments or return other functions.

This level of modularity is allowed because functions are first-class objects in JavaScript.

```
function fun() {
    console.log("Hello, World!");
}
function fun2(action) {
    action();
}

fun2(fun);
```

Array Methods

There are some very helpful array methods that are higher order functions:

- `forEach(callbackFn)`
- `map(callbackFn)`
- `filter(callbackFn)`
- `find(callbackFn)`

Each of their arguments take in **synchronous callback functions** that execute on each element of the array.

Callback Function Syntax

Most callback functions, especially in array methods, are written with arrow functions:

```
names.forEach(name => console.log(name));
let upper = names.map(name => name.toUpperCase());
```

```
let numbers = [3, 7, 6, 9];
let result = numbers
  .map(n => n * 2) // [6, 14, 12, 18]
  .filter(n => n > 10); // [14, 12, 18]
console.log(result);
```

Array Methods: forEach

`forEach(callbackFn)`: Executes a callback function on each array item. The return value is discarded.

```
let names = ["Ada", "Jay", "Michael"];

// foreach
names.forEach((name, i) => console.log(`Index ${i}: ${name}`));

// loop
for (let i = 0; i < names.length; i++) {
  console.log(`Index ${i}: ${names[i]}`);
}
```

Penn Engineering

Array Methods: map

`map(callbackFn)`: Creates a new array populated by calling a function on each element of the given array

```
let pricesInCents = [1499, 2599, 350, 4999];

let formattedPrices = pricesInCents.map(cents => {
  let dollars = (cents / 100).toFixed(2);
  return `$${dollars}`;
});

console.log(formattedPrices);
// [ '$14.99', '$25.99', '$3.50', '$49.99' ]
```

Array Methods: filter

`filter(callbackFn)`: Creates a copy of the given array filtered down to only elements that pass the condition implemented in the function (function must return truthy)

```
let odds = [1, 2, 3, 4, 5].filter(num => num % 2 !== 0); // [1, 3, 5]

let users = [
  { name: "Gonzalo", emailVerified: true },
  { name: "Trip", emailVerified: false },
  { name: "Grace", emailVerified: true }
];

let verifiedUsers = users.filter(user => user.emailVerified);

console.log(verifiedUsers);
// [ { name: Gonzalo, emailVerified: true }, { name: 'Grace',
emailVerified: true } ]
```

Array Methods: find

`find(callbackFn)`: Returns the first element in the given array that satisfies the condition in the callback function

```
let words = ["hi", "hello", "hey"];
let longWord = words.find(w => w.length > 2); // "hello"
```

Penn Engineering

Array Methods: some and every

`some(callbackFn)`: Returns true if any element passes the function's condition

`every(callbackFn)`: Returns true only if if all elements pass the function's condition

```
let hasNegative = [1, -2, 3].some(x => x < 0); // true
let allPositive = [1, 2, 3].every(x => x > 0); // true
```

POLL QUESTION

```
const nums = [1, 2, 3, 4, 5, 6];
```

Given this array, choose the expression that will produce a new array with only the even numbers from the original array, and those numbers are doubled.

```
nums.filter(n => n % 2 ===
0).map(n => n * 2)
```

or

```
nums.map(n => n % 2 === 0 ? n * 2
: n).filter(n => n % 2 === 0)
```

POLL QUESTION

```
const names = ["anna anna", "Bob",
"Eve", "Otto", "Ada"];
```

Which of these expressions returns the first index of a palindrome (same name spelt forwards and backwards)?

```
names.findIndex(name =>
name.toLowerCase() ===
name.toLowerCase().split('').rever
se().join(''));
```

Live Coding: String Methods