Collections, Control Flow, and Strings

CIS 1962 (Winter 2026)
January 22nd, 2026

Penn Engineering

Lesson Plan

More Collections

5	Maps & Sets
8	Destructuring, Spread, Rest

Higher Order Functions

17	Array Methods

Control Flow

30	Conditional Statements
41	Loops

Strings

50	Array Methods

More Collections

Review Activity

https://edstem.org/us/courses/91614/lessons/157464/slides/926372

Turn:

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

Into:

```
arr = [1, 'a', 'b', 4, 5]
x = [2, 3, 4]
y = [2, 3]
```

Solution:

```
x = arr.slice(1, 4);
y = arr.splice(1, 2, 'a', 'b');
```

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

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

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:

expanding

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

combining

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

copying

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

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

Coding Activity

https://edstem.org/us/courses/91614/lessons/157464/slides/926593

Extract the title, width, and theme from the following object:

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

Answer:

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

Coding Activity

https://edstem.org/us/courses/91614/lessons/157464/slides/926593

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

Given the following objects, use only spread and object syntax merge them into

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

Answer:

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

Coding Activity

https://edstem.org/us/courses/91614/lessons/157464/slides/926593

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

Given the following object, use only rest and object syntax to extract
`{ role: "admin", meta: { active: true } }`

Answer:

```
const { id, name, email,
...rest } = user;
```

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]}`);
}
```

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"
```

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
```

Coding Activity

Manipulate the array below to extract only the even numbers, then double those numbers.

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

Answer:

```
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)
```

5-Minute Break!

Control Flow

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

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 (isButtonActive)
```

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

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

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

```
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
}
```

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

Nesting conditions, inserting dynamic content into HTML

Coding Activity

```
let result = ""
let userStatus = "pending"
switch (userStatus) {
    case "active":
        result = "User is active";
        break;
    case "inactive":
        result = "User needs activation";
        break;
    case "banned":
        result = "User is banned";
        break;
    case "deleted":
        result = "User not found";
        break;
    default:
        result = "Pending or Unknown status";
        break;
}
```

```
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"
                : "Pending or Unknown status";
```

https://edstem.org/us/courses/91614/lessons/157464/slides/926981

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;
     }
}
```

Coding Activity

Write a function, sumDigits, that uses a for of loop to sum the digits of a number passed as an argument.

https://edstem.org/us/courses/91614/lessons/157464/slides/926992

Answer:

```
function sumDigits(num) {
    let sum = 0;
    for (const digit of String(num)) {
        sum += digit - '0';
    }
    return sum;
}
```

Strings

Strings

JavaScript Strings behave like objects when you use string methods, but are primitives.

Strings are considered **immutable sequences of UTF-16 code units**.

```
"abc".length    // \u0061\u0062\u0063
"🙂".length     // \ud83d\ude42
```

Iterating Over Strings

You can use brackets [] to read characters by index.

```
let string = “hello world”
console.log(string[0]) // h
```

As a sequence, you may use a for loop to iterate through strings. However, because of UTF-16 encoding, index-based loops sometimes break.

```
let emojis = "🙂🙂🙂"

// this prints 6 times!
for (let i = 0; i < emojis.length; i++) {
    console.log(emojis[i])
}
```

Iterating Over Strings

Instead try using a for of loop:

```
let emojis = "🙂🙂🙂"

for (let c of emojis) {
    console.log(c)
}
```

String Methods

There are many string methods you can use for string manipulation:

- charAt
- concat
- includes, startWith, endsWith
- indexOf, lastIndexOf
- slice, substring, replace, replaceAll, split
- toLowercase

Examples of these are on the Ed Lessons page:

https://edstem.org/us/courses/91614/lessons/157464/slides/927333

Coding Activity

https://edstem.org/us/courses/91614/lessons/157464/slides/927868

Given an array of strings, filter out the palindromes.

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

Answer:

```
let palindromes = names.filter(name =>
name.toLowerCase() ===
name.toLowerCase().split('').reverse().join(''));
```

Looking Ahead: JS Modules and Libraries

JavaScript is Vast

JavaScript is a well-supported language with many external libraries and modules for development and web page functionality.

We've covered a lot of the basic syntax and functionality of the JS language in the last 2 lectures, so in the next lecture, we will go over how you add more functionality and libraries to JavaScript, including:

- Style Checking and Formatting (eslint, prettier)
- TypeScript (superset of JS)
- Custom modules (import/export scripts)