COMP1531

🛠 Development - Javascript

Lecture 1.2

Author(s): Hayden Smith, Dr. Yuchao Jiang

In This Lecture

- **Why?** 🤔
  - Javascript is a valuable tool to learn and necessary for the project
- **What?** 📰
  - Learning a second language
  - Javascript vs C
  - Core javascript language features

🤟 Javascript

- Popularity and Demand
  - Job Market: JavaScript is one of the most in-demand programming languages.
  - Industry Relevance: Major companies like Google, Facebook, and Netflix use JavaScript extensively.
- Web Development
  - JavaScript is the de facto language for client-side web development. JavaScript can be used for both front-end and back-end development.
- Rapid Prototyping
  - Javascript has an extremely rich open source library and package manager that allows you to build apps quickly. Javascript is very high level, making it easy to write code.

🤟 Javascript

examples:

- Clickable Button that Changes Text
- Simple Form Validation
- Fetching and Displaying Data from an API

👩🏻‍💼 Disclaimer

- Because you already know C, we will teach Javascript very quickly and mainly focus on the differences between Javascript and C.
- Unlike C, Javascript has a sprawling set of capabilities - the language will feel much bigger, and therefore you might feel you have a poorer grasp on it.
- Don't expect to know everything about Javascript this term. Just focus on only learning what you need to solve a problem at hand, and you will learn more super quick.

🤟 Javascript

```
const z = 3;
function hello(a, b, c) {
  return `${a} ${b}`;
}
```

intro.js

🧑‍🏫 Learning Another Language

In the case of learning another language like Javascript after doing COMP1511 with C, the main hurdles we have to overcome are:

- Javascript does not have programmer-defined types, unlike C
- Javascript has object-oriented components (which we can somewhat ignore), unlike C
- Javascript does not deal with pointers, unlike C (yay)
- Javascript is often written at a "higher level" (more abstract)
- Javascript does not have an intermediate compilation step, like C

🤺 Javascript VS C

Write a function that takes in two numbers, and returns the smaller number

C

```c
int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}
```

compare_1.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}
```

compare_1.js

🤺 Javascript VS C

Write a function that takes in two numbers, and returns the smaller number

C

```c
int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}
```

compare_1.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}
```

compare_1.js

Now let's call the function and print the result!

🤺 Javascript VS C

Write a function that takes in two numbers, and returns the smaller number

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

🤺 Javascript VS C

Write a function that takes in two numbers, and returns the smaller number

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

Now let's run the program

🤺 Javascript VS C

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

```
gcc -Wall -Werror -O -o 1.4_compare_2.c -o runnable
./runnable
```

```
node 1.4_compare_2.js
```

🤺 Javascript VS C

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

```
gcc -Wall -Werror -O -o 1.4_compare_2.c -o runnable
./runnable
```

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

```
node 1.4_compare_2.js
```

OK but:

🤺 Javascript VS C

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

```
gcc -Wall -Werror -O -o 1.4_compare_2.c -o runnable
./runnable
```

```
node 1.4_compare_2.js
```

OK but:

What is node? 😵‍💫

🤺 Javascript VS C

C

```c
#include <stdio.h>

int minimum(int a, int b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

int main(int argc, char* argv[]) {
  printf("%d\n", minimum(3, 5));
}
```

compare_2.c

```
gcc -Wall -Werror -O -o 1.4_compare_2.c -o runnable
./runnable
```

Javascript

```javascript
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}

console.log(minimum(3, 5));
```

compare_2.js

```
node 1.4_compare_2.js
```

OK but:

What is node? 😵‍💫

Are there steps missing? 😰

🥳 NodeJS

NodeJS is a command line interface that interprets Javascript code within a runtime environment that is built on Google's V8 engine. 😵‍💫

Or if you want a simpler explanation...

NodeJS is the program that compiles and runs Javascript.

To really oversimplify it, NodeJS has a similar function to GCC.

🥳 NodeJS

- **NodeJS** is what's known as an **interpreted** language instead of a **compiled** language. This means that the program is compiled and run as part of the same *step*.

This has two implication:

- A little slower to run, because it has to compile to runnable code every time.
- A little more convenient, as changes to code don't require an extra compilation step.

🤺 Performance V Convenience

But let's go and learn more about the language...

🦄 Variables, Printing

Variables are containers for data values.

`const`, `let`, `console.log`

```javascript
// Variables declared with "let"
//  can be modified after definition
let years = 5;

// Variables declared with "const"
//  cannot be modified after definition
const fullname = 'Giraffe';
const age = 18;
const height = 2048.11;
const notexist = undefined;
const existbutnothing = null;

// You print with console.log
console.log(years);
console.log(fullname);
console.log(height);

// Double and single apostrophes are equivalent
```

```
console.log('Hello!');
console.log("how are you?");
```

variables.js

🦄 Strings

Concatenation, string literals

```js
// We can easily join strings together!
let sentence = 'My';
sentence = sentence + ' name is';
sentence += ' Pikachu';
console.log(sentence);

// If you need to mix variables and
//  strings, you can create a string literal
const age = 7;
const fullname = 'Yuchao';
const phrase = `Hello! My name is ${fullname} and I am ${age}`;
console.log(phrase);
```

strings.js

Literals provide an easy way to interpolate variables and expressions into strings. The method is called string interpolation.

🦄 Control Structures

`if`, `else if`, `else`, `while`, `for`.

```javascript
const number = 5;
if (number > 10) {
  console.log('Bigger than 10');
} else if (number < 2) {
  // Do nothing
} else {
  console.log('Number between 2 and 9');
}

console.log('--------------------------');

let i = 0;
while (i < 5) {
  console.log('Hello there');
  i += 1;
}

console.log('--------------------------');

for (let i = 0; i < 5; i++) {
  console.log('Hello there');
}
```

control_structures.js

🦄 Functions

Very similar syntax to C

```
function minimum(a, b) {
  if (a > b) {
    return b;
  } else {
    return a;
  }
}
```

compare_1.js

Pause for a bit of theory...

📦 Data Structures: Collections

We'll now discuss two important data structures that are both **collections** of data.

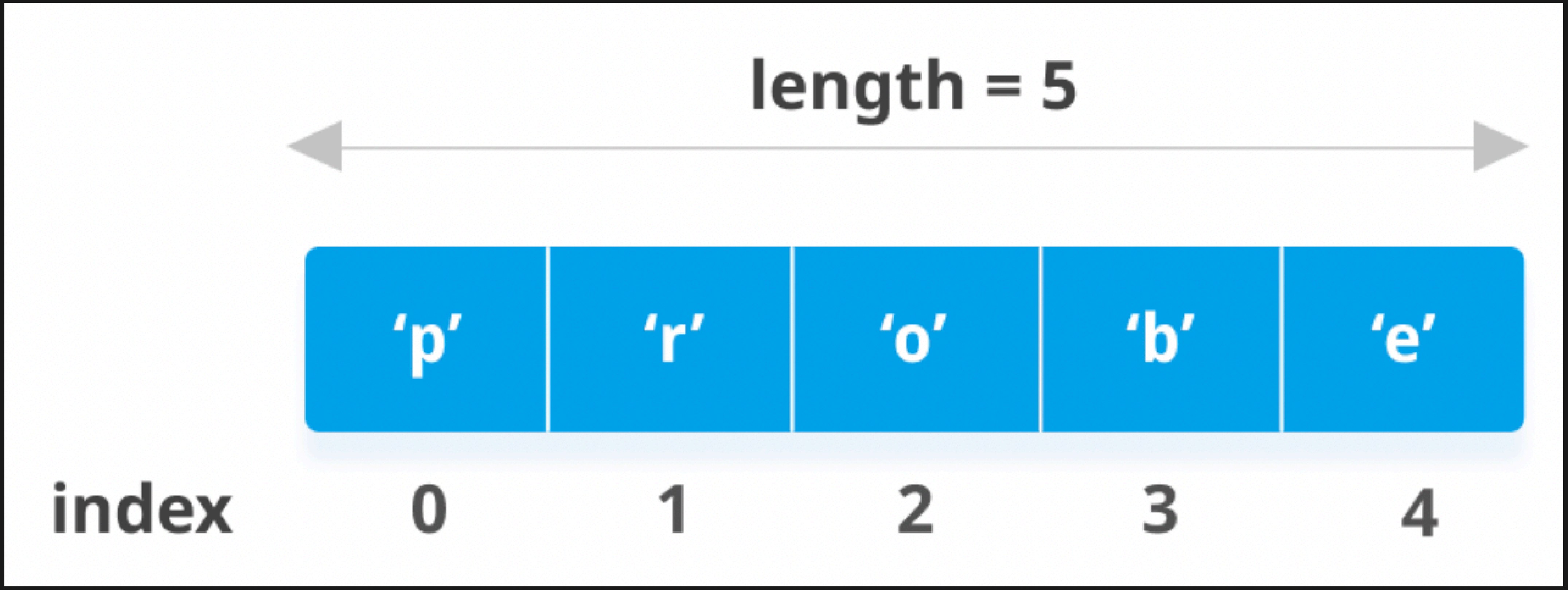
Collections can either be:

- Sequential collections
- Associative collections

🔢 Sequential Collections

In **sequential collections** values are referenced by their integer index (key) that represents their location in an order.

In Javascript sequential collections are represented by an **array**.



🗺️ Associative Collections

In **associative collections** values are referenced by their string key that maps to a value. They often do not have an inherent sense of order.

- `name` → `"sally"`
- `age` → `18`
- `height` → `"187cm"`

Unpause, back to code!

🍋 Arrays

An array is a special variable. An array can hold many values under a single name, and you can access the values by referring to an index number. Arrays are dynamic and can hold different data types.

```js
// This is a array
const names = ['Hayden', 1, 'Darcy', 'Giuliana'];

console.log(`1 ${names}`);
console.log(`2 ${names[0]}`);
names[1] = 'Yuchao';
names.push('Rani');
console.log(`3 ${names}`);

console.log(names.length);
```

arrays.js

names.push? -> Array Methods

🍋 Arrays

Iterate through an array.

```javascript
const items = ['a', 'b', 'c', 'd', 'e'];

let i = 0;
while (i < 5) {
  console.log(items[i]);
  i++;
}

for (let j = 0; j < 5; j++) {
  console.log(items[j]);
}

for (let k = 0; k < items.length; k++) {
  console.log(items[k]);
}
```

array_basic.js

🍋 Arrays

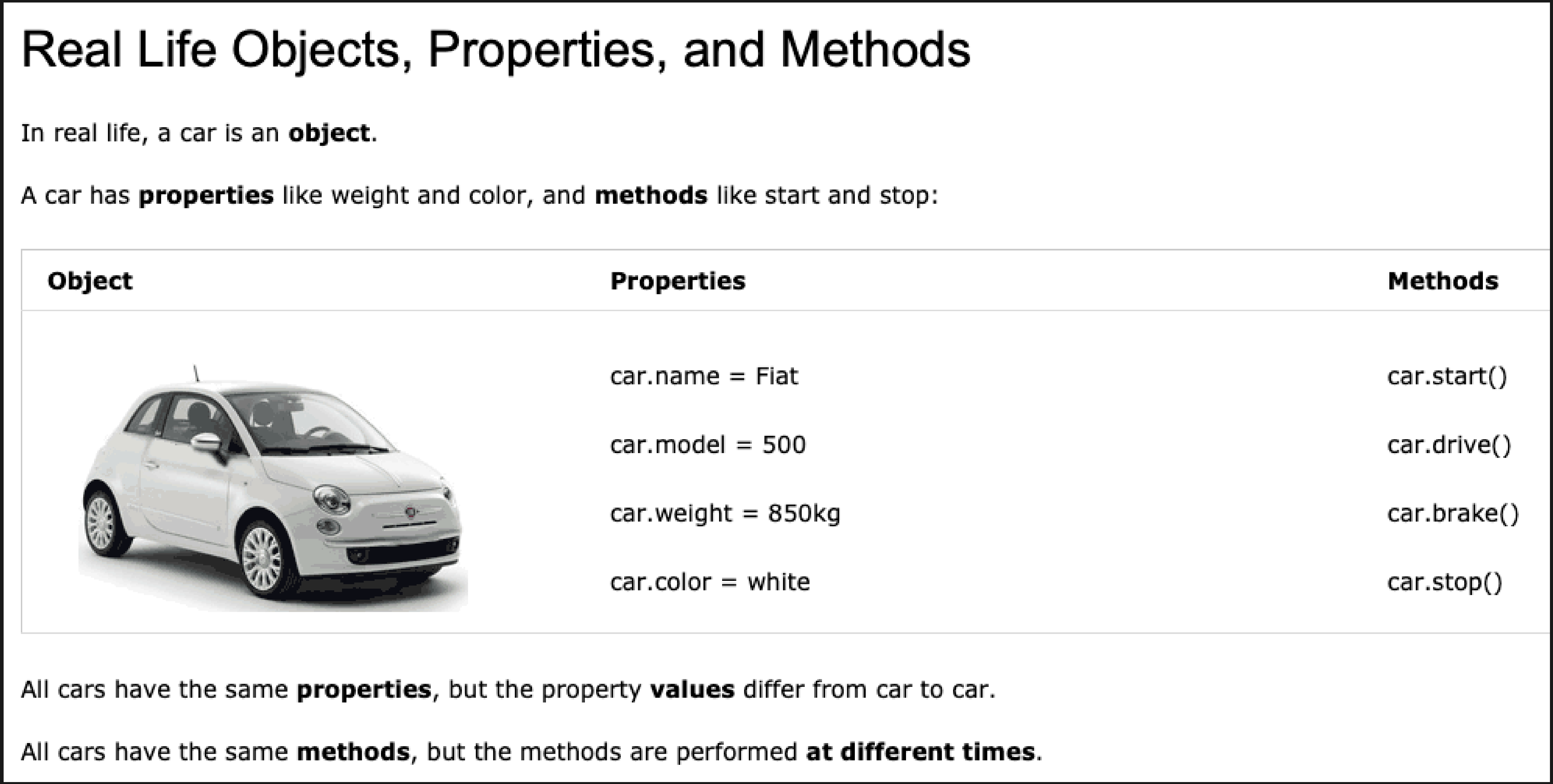
For In Loop v.s. For Of Loop

```
const items = ['a', 'b', 'c', 'd', 'e'];

// prints 0, 1, 2, 3, 4
for (const i in items) {
  console.log(items[i]);
}

// prints a, b, c, d, e
for (const item of items) {
  console.log(item);
}

console.log(items.includes('c'));
```

array_advanced.js

- For In statement loops through the properties of an array.
- For Of statement loops through the values of an array.

Objects



Real Life Objects, Properties, and Methods

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:

Object	Properties	Methods

	car.name = Fiat
	car.model = 500
	car.weight = 850kg
	car.color = white
		car.start()
		car.drive()
		car.brake()
		car.stop()

All cars have the same **properties**, but the property **values** differ from car to car.

All cars have the same **methods**, but the methods are performed **at different times**.

source

🍊 Objects

- Objects are variables too. But objects can contain many values.
- This code assigns many values (Fiat, 500, white) to a variable named car:

```javascript
const car = {type:"Fiat", model:"500", color:"white"};
```

- The values are written as name:value pairs (name and value separated by a colon).
- It is a common practice to declare objects with the const keyword.

🍊 Objects

- Objects are associative structures that may consist of many different types.
- You can use them when you need a collection of items that are identified by a string description, rather than a numerical index (arrays).

```js
const student = {
  name: 'Emily',
  score: 99,
  rank: 1,
};

console.log(student);
console.log(student.name);
console.log(student.score);
console.log(student.rank);

student.height = 159;
console.log(student);
```

objects.js

🍊 Objects

We can create and populate objects different ways.

```js
const userData = {};
userData.name = 'Sally';
userData.age = 18;
userData.height = '187cm';
console.log(userData);
```

object_basic1.js

```js
const userData = {
  name: 'Sally',
  age: 18,
  height: '187cm',
};
console.log(userData);
```

object_basic2.js

Both of these programs would print `{ name: 'Sally', age: 18, height: '187cm' }`

🍊 Objects

We can mix the two methods, and also use alternative syntax with assigning.

```javascript
userData.prop = 1;
userData['prop'] = 1;
```

Or in a more full example.

```javascript
// You can assign more keys even
//  after creation
const userData = {
  name: 'Sally',
  age: 18,
};
userData.height = '187cm';

console.log(userData);
```

object_more1.js

```javascript
// You can reference keys with either
//  obj.key or obj['key']
const userData = {};
userData.name = 'Sally';
userData.age = 18;
userData.height = '187cm';
console.log(userData);
```

object_more2.js

🍊 Objects

We can also get various properties of an object using the `Object` functions.

```javascript
const userData = {
  name: 'Sally',
  age: 18,
  height: '187cm',
};

const keys = Object.keys(userData);
const entries = Object.entries(userData);
const values = Object.values(userData);

console.log(keys);
console.log(entries);
console.log(values);
```

object_props_1.js

```
[ 'name', 'age', 'height' ]
[ [ 'name', 'Sally' ], [ 'age', 18 ], [ 'height', '187cm' ] ]
[ 'Sally', 18, '187cm' ]
```

🧐 Further Discussion Of Objects

The following code exhibits behavior you're probably not used to:

```javascript
const arr = [1, 2, 3];
console.log(arr.length);
console.log(arr.includes(3));
```

object_model.js

"arr" is an array, but it also seems to have:

- A property `length` that we never set?
- Some kind of function that is being called?

Let's look at why this is.

🍅 In JavaScript, Almost "Everything" Is An Object.

- This array is an object.
- An "object" being a data type that:
  - Contains 0 or more properties (/attributes)
  - Contains 0 or more functions (/methods)

👯 Tying Some Things Together

Let's try some lists of objects (an array of objects).

```
const userData = [
  {
    name: 'Sally',
    age: 18,
    height: '186cm',
  }, {
    name: 'Bob',
    age: 17,
    height: '188cm',
  },
];

// Returns an array of the object's
// own enumerable property names.
const keys = Object.keys(userData);
console.log(keys);

// Returns an array of the object's own
// enumerable property [key, value] pairs.
const entries = Object.entries(userData);
console.log(entries);
```

```javascript

// Returns an array of the object's own
// enumerable property values.
const values = Object.values(userData);
```

object_loop1.js

👯 Tying Some Things Together

Let's try some lists of objects (an array of objects).

```
const userData = [
  {
    name: 'Sally',
    age: 18,
    height: '186cm',
  }, {
    name: 'Bob',
    age: 17,
    height: '188cm',
  },
];

for (let i = 0; i < userData.length; i++) {
  console.log(`${userData[i].name}'s properties are:`);
  console.log(`  name: ${userData[i].name}`);
  console.log(`  age: ${userData[i].age}`);
  console.log(`  height: ${userData[i].height}`);
}
```

object_loop2.js

👯 Tying Some Things Together

Let's try some lists of objects.

```javascript
const userData = {
  Sally: {
    age: 18,
    height: '186cm',
  },
  Bob: {
    age: 17,
    height: '188cm',
  },
};

for (const key in userData) {
  console.log(`${key}'s properties are:`);
  for (const key2 in userData[key]) {
    console.log(`  ${key2}: ${userData[key][key2]}`);
  }
}
```

object_loop3.js

👯 Tying Some Things Together

Let's try more lists of objects.

```
const student1 = { name: 'Yuchao', score: 50 };
const student2 = { name: 'Rani', score: 91 };
const student3 = { name: 'Hayden', score: 99 };
const students = [student1, student2, student3];

console.log(students);

// Approach 1
const numStudents = students.length;
for (let i = 0; i < numStudents; i++) {
  const student = students[i];
  if (student.score >= 85) {
    console.log(`${student.name} got an HD`);
  }
}

// Approach 2
for (const student of students) {
  if (student.score >= 85) {
    console.log(`${student.name} got an HD`);
  }
}
```

combining.js

📕 Further Reading

- array of objects

👂 Feedback

Or go to the form here.