short-circuit evaluation

Two important aspects of logical operators in JavaScript, is that they evaluate from left to right, and they **short-circuit**.

```
var person = {
  name: 'Jack',
  age: 34
}console.log(person.job || 'unemployed');
// prints 'unemployed'
```

The list of *falsy* values includes:

- 0
- Empty strings like "" or ''
- null which represents when there is no value at all
- undefined which represents when a declared variable lacks a value
- NaN, or Not a Number

Hoisting Feature in JavaScript

MDN Web Docs: JavaScript **Hoisting** refers to the process whereby the interpreter allocates memory for variable and function declarations prior to execution of the code.

We should also be aware of the *hoisting* feature in JavaScript which allows access to function declarations before they're defined. But it's not a good practice!

Take a look at an example of hoisting:

```
greetWorld(); // Output: Hello, World!

function greetWorld() {
  console.log('Hello, World!');
}
```

JavaScript only hoists declarations, not initializations. If a variable is used in code and then declared and initialized, the value when it is used will be its default initialization (`undefined` for a variable declared using `var`, otherwise uninitialized). For example:

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)

var num; // Declaration

num = 6; // Initialization
```

The example below only has initialization. No hoisting happens so trying to read the variable results in `ReferenceError` exception.

```
console.log(num); // Throws ReferenceError exception

num = 6; // Initialization
```

NOTE: In JavaScript, **all function arguments are always passed by value**.

Default Parameters

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is `undefined` when called.

Take a look at the code snippet below that uses a default parameter:

```
function greeting (name = 'stranger') {
  console.log(`Hello, ${name}!`)
}

greeting('Nick') // Output: Hello, Nick!
greeting() // Output: Hello, stranger!
```

Take a look at the code snippet below:

```
function makeShoppingList(item1='milk', item2='bread', item3='eggs'){
  console.log(`Remember to buy ${item1}`);
  console.log(`Remember to buy ${item2}`);
  console.log(`Remember to buy ${item3}`);
}

makeShoppingList('milk', 'water');
```

outputs:

```
Remember to buy milk
Remember to buy water
Remember to buy eggs
```

Functions Expressions

Another way to define a function is to use a *function expression*. To define a function inside an expression, we can use the `function` keyword. In a function expression, the function name is usually omitted. A function with no name is called an *anonymous function*. A function expression is often stored in a variable in order to refer to it.

Consider the following function expression:

```
const calculateArea = function(width, height) {

     const area = width * height;

     return area;
```

Since the release of ES6, it is common practice to use `const` as the keyword to declare the variable. We use `const` to protect this function; so it will never be deleted or changed.

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

What should we do now? Express functions or Declare them?

Similar to the `var` statement, function declarations are hoisted to the top of other code. Function expressions aren't hoisted, which allows them to retain a copy of the local variables from the scope where they were defined.

Benefits of Function Expressions

There are several different ways that function expressions become more useful than function declarations.

- As closures
- As arguments to other functions
- As Immediately Invoked Function Expressions (IIFE)

As we've seen, function expressions don't offer much that can't be achieved with function declarations, but using them can often result in cleaner and more readable code.

Concise Body Arrow Functions

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We'll explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required

2. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow `=>` and the `return` keyword can be removed. This is referred to as *implicit return*.

Conclusion About Functions

We have 3 different ways to create functions if JavaScript:

1. Function Declaration
2. Function Expression
3. Arrow Function

Make sure you know about their use and pros and cons

Scope Pollution

```
let num = 50;

const logNum = () => {
  num = 100; // Take note of this line of code
  console.log(num);
};

logNum(); // Prints 100
console.log(num); // Prints 100
```

You'll notice:

- We have a variable `num`.
- Inside the function body of `logNum()`, we want to declare a new variable but forgot to use the `let` keyword.
- When we call `logNum()`, `num` gets reassigned to `100`.
- The reassignment inside `logNum()` affects the global variable `num`.
- Even though the reassignment is allowed and we won't get an error, if we decided to use `num` later, we'll unknowingly use the new value of `num`.

Arrays

If you try to reach an out of bound index of an array, you'll get `undefined` (either -1 or less or bigger than the size of the array).

You can update elements inside an array that has been declared using both `let` and `const` keywords. But you can only change the whole array (assign it to a whole new array) if it has been declared by `let` keyword beforehand!

Array functions and properties

- `length` property
- `push()` function takes one or more parameters and adds them to the end; it then returns the length of the current array.
- `pop()` function removes the last element from an array and returns it.
- `shift()` function removes the first element from an array and returns it.
- `unshift()` function add one or more parameters to the start of an array.
- `slice()` function take zero, one or more parameters and slices the array off from the first param till the end or till the second param and returns a copy of the sliced array (Note that `slice()` is non-mutating).
- `splice(pos, n)` function removes n elements from the pos index of an array and returns the removed items.

More about `slice()`:

```
slice() copies the whole array

slice(start) [start, array.length-1]

slice(start, end) [start, end - 1]
```

```
const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));

// expected output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4));

// expected output: Array ["camel", "duck"]

console.log(animals.slice(-2));

// expected output: Array ["duck", "elephant"]

console.log(animals.slice(2, -1));

// expected output: Array ["camel", "duck"]
```

More about these functions:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Pass by Ref or Value?

In JavaScript, objects and arrays are passed by reference to functions; whereas other data types - numbers for example – are pass by value.

Nested Array (nD array)

```
const nestedArr = [[1], [2, 3]];
console.log(nestedArr[1]); // Output: [2, 3]
console.log(nestedArr[1][0]); // Output: 2
```

Functions as Data types

Take a look at the code snippet below:

```
const announceThatIAmDoingImportantWork = () => {
    console.log("I'm doing very important work!");
};
```

You can see that the name of our function is so long. There's a solution for that, since functions are *first class objects* in JavaScript.

```
const busy = announceThatIAmDoingImportantWork;
busy(); // This function call barely takes any space!
```

`busy` is a variable that holds a *reference* to our original function. If we could look up the address in memory of `busy` and the address in memory of `announceThatIAmDoingImportantWork` they would point to the same place. Our new `busy()` function can be invoked with parentheses as if that was the name we originally gave our function.

Note: if you log busy.name to the console, you will get the original name of the function!

A question remains unanswered here: Why???

1. First is that it enables us to pass functions as parameters to other functions.
2. Second and finally, there are cases where we write something like `function1 = function2` to use the name `function1` as a synonym for `function2` when the name `function1` is for some reason easier to type/remember than `function2`.

Functions as Parameters

A *higher-order function* is a function that either accepts functions as parameters, returns a function, or both!

We call the functions that get passed in as parameters and invoked *callback functions* because they get called during the execution of the higher-order function.

Iterators

Let's go through some high order functions (HOF) for iterating an array!

- **forEach**

You can either pass a normal function or an arrow function to forEach iterator method or a predefined function (second code snippet). Note that this method will always return undefined.

```
groceries.forEach(groceryItem => console.log(groceryItem));

function printGrocery(element){
  console.log(element);
}
groceries.forEach(printGrocery);
```

- map

When `.map()` is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling `.map()`:

```
const numbers = [1, 2, 3, 4, 5];
const bigNumbers = numbers.map(number => {
  return number * 10;
});
```

Notice that the first array (numbers) remained the same!

- filter

Like `.map()`, `.filter()` returns a new array. However, `.filter()` returns an array of elements after filtering out certain elements from the original array. The callback function for the `.filter()` method should return `true` or `false` depending on the element that is passed to it. The elements that cause the callback function to return `true` are added to the new array.

```
const words = ['chair', 'music', 'pillow', 'brick', 'pen',
'door'];
const shortWords = words.filter(word => {
  return word.length < 6;
});
```

- findIndex

Do not mistake this method with charAt(index)!

```
const jumbledNums = [123, 25, 78, 5, 9];
const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
```

If there isn't a single element in the array that satisfies the condition in the callback, then `.findIndex()` will return `-1`.

- **Reduce**

Another widely used iteration method is `.reduce()`. The `.reduce()` method returns a single value after iterating through the elements of an array, thereby *reducing* the array. Take a look at the example below:

```
const numbers = [1, 2, 4, 10];
const summedNums = numbers.reduce((accumulator, currentValue)
=> {
  return accumulator + currentValue
})
console.log(summedNums) // Output: 17
```

Before we jump into explaining how each line is being executed, what does this code snipped mainly does??? It's adding all the elements inside the array which is 17 at last.

Now let's go over the use of `.reduce()` from the example above:

- `numbers` is an array that contains numbers.
- `summedNums` is a variable that stores the returned value of invoking `.reduce()` on `numbers`.
- `numbers.reduce()` calls the `.reduce()` method on the `numbers` array and takes in a callback function as argument.
- The callback function has two parameters, `accumulator` and `currentValue`. The value of `accumulator` starts off as the value of the first element in the array and the `currentValue` starts as the second element. To see the value of `accumulator` and `currentValue` change, review the chart above.
- As `.reduce()` iterates through the array, the return value of the callback function becomes the `accumulator` value for the next iteration, `currentValue` takes on the value of the current element in the looping process.

The `.reduce()` method can also take an optional second parameter to set an initial value for `accumulator` (remember, the first argument is the callback function!). For instance:

```
const numbers = [1, 2, 4, 10];
const summedNums = numbers.reduce((accumulator, currentValue)
=> {
  return accumulator + currentValue
}, 100)  // <- Second argument for .reduce()
console.log(summedNums); // Output: 117
```

Real world usage of .reduce()

- Factorial

```
[1,2,3,4,5,6,7].reduce((a, b) => a * b)    // 5040
```

- Join strings

```
array = ['A', 'quick', 'brown', 'fox', 'jumps', 'over',
'the', 'lazy', 'dog'];
array.reduce((a, b) => a + ' ' + b)
// "A quick brown fox jumps over the lazy dog"
```

- Collecting info from an html form

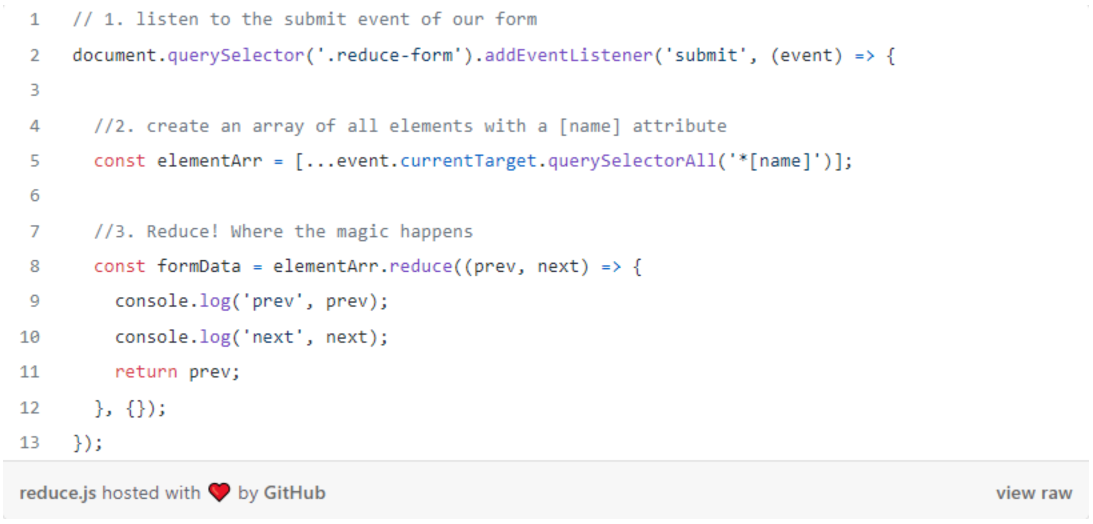
Our real world use case is going to collect input values from a form and send them to a backend api.

```
<form class="reduce-form">

  <fieldset>

    <input type="text" name="firstname" placeholder="Enter first name" /><br/>

    <input type="text" name="lastname" placeholder="Enter last name"><br/>

    <input type="email" name="email" placeholder="Enter e-mail" /><br/>

    <button type="submit">Submit</button>

  </fieldset>

</form>
```

And we’re going to create a function with `reduce` that goes through the DOM of the form and creates a nice Object that we can send to our api! The Object will have the following shape:

```
{
firstname: value,
lastname: value,
email: value
}
```

And in our JavaScript we have:

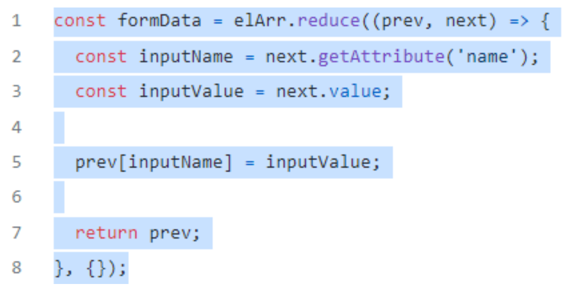


```javascript
// 1. listen to the submit event of our form
document.querySelector('.reduce-form').addEventListener('submit', (event) => {

  //2. create an array of all elements with a [name] attribute
  const elementArr = [...event.currentTarget.querySelectorAll('*[name]')];

  //3. Reduce! Where the magic happens
  const formData = elementArr.reduce((prev, next) => {
    console.log('prev', prev);
    console.log('next', next);
    return prev;
  }, {});
});
```

If we log the above html and javascript to our console:

```
"prev", Object {}
"next", "<input type='text' name='firstname' placeholder='Enter first name'>"
"prev", Object {}
"next", "<input type='text' name='lastname' placeholder='Enter last name'>"
"prev", Object {}
"next", "<input type='email' name='email' placeholder='Enter e-mail'>"
```

Now, let's build our object! First we get the name and value from `next`. The next step is to update our `prev` Object and return it, so our next iteration has the new data available.



```javascript
const formData = elArr.reduce((prev, next) => {
  const inputName = next.getAttribute('name');
  const inputValue = next.value;

  prev[inputName] = inputValue;

  return prev;
}, {});
```

We're dynamically setting the name of our key/value pair with brackets and returning the updated `prev` value, so our next iteration can use it. When it's done looping, the `const formData` will have its final value and contain all the names of our inputs with its corresponding values. After that you can use it to do amazing things. Like sending it your backend api, or whatever.

- Some --- Array.prototype.some()

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function. It returns true if, in the array, it finds an element for which the provided function returns true; otherwise it returns false. It doesn't modify the array.

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

// checks whether an element is even

const even = (element) => element % 2 === 0;

console.log(array.some(even));

// expected output: true
```

// Arrow function

some((element) => { ... })

some((element, index) => { ... })

some((element, index, array) => { ... })

// Callback function

some(callbackFn)

some(callbackFn, thisArg)

// Inline callback function

some(function callbackFn(element) { ... })

some(function callbackFn(element, index) { ... })

some(function callbackFn(element, index, array){ ... })

some(function callbackFn(element, index, array) { ... }, thisArg)

- **Every --- Array.prototype.every()**

Exactly like some, but returns true only if the condition – or let’s better say “the function” passed to it is satisfied by ALL THE ELEMENTS inside the array.

Overview

A class property can be accessed via the dot operator or brakcets. With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions.

We **must** use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

```
let spaceship = {
  'Fuel Type': 'Turbo Fuel',
  'Active Duty': true,
  homePlanet: 'Earth',
  numCrew: 5
};
spaceship['Active Duty'];   // Returns true
spaceship['Fuel Type'];   // Returns  'Turbo Fuel'
spaceship['numCrew'];   // Returns 5
spaceship['!!!!!!!!!!!!!!!'];   // Returns undefined
```

```
let returnAnyProp = (objectName, propName) =>
objectName[propName];

returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'
```

Property Assignment

One of two things can happen with property assignment:

- If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
- If there was no property with that name, a new property will be added to the object.

It's important to know that although we can't reassign an object declared with `const`, we can still mutate it, meaning we can add new properties and change the properties that are there.

```
const spaceship = {type: 'shuttle'};
spaceship = {type: 'alien'}; // TypeError: Assignment to
constant variable.
spaceship.type = 'alien'; // Changes the value of the type
property
spaceship.speed = 'Mach 5'; // Creates a new key of 'speed'
with a value of 'Mach 5'
```

You can delete a property from an object with the `delete` operator.

```
const spaceship = {
  'Fuel Type': 'Turbo Fuel',
  homePlanet: 'Earth',
  mission: 'Explore the universe'
};

delete spaceship.mission;  // Removes the mission property
```

Methods

We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method's name, while the value is an anonymous function expression.

```
const alienShip = {
  invade: function () {
    console.log('Hello! We have come to dominate your
planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```

With the new method syntax introduced in ES6 we can omit the colon and the `function` keyword.

```
const alienShip = {
  invade () {
    console.log('Hello! We have come to dominate your
planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```

Nested Objects

We can chain operators to access nested properties. We'll have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

```
spaceship.nanoelectronics['back-up'].battery;
// or
const capFave = spaceship.crew.captain ['favorite foods'][0];
```

Pass by Reference

Objects are *passed by reference*. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a `const` variable).

```
const spaceship = {
  homePlanet : 'Earth',
  color : 'silver'
};

let paintIt = obj => {
  obj.color = 'glorious gold'
};

paintIt(spaceship);

spaceship.color // Returns 'glorious gold'
```

Let’s take a look at an example:

```
et spaceship = {
  'Fuel Type' : 'Turbo Fuel',
  homePlanet : 'Earth'
};

let greenEnergy = obj => {
  obj['Fuel Type'] = 'avocado oil';
};

let remotelyDisable = obj => {
  obj.disabled = true;
};

greenEnergy(spaceship);
remotelyDisable(spaceship);
console.log(spaceship);
```

Output:

```
{ 'Fuel Type': 'avocado oil',
  homePlanet: 'Earth',
  disabled: true }
```

Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects aren’t ordered! JavaScript has given us alternative solution for iterating through objects with the `for...in` syntax .

`for...in` will execute a given block of code for each property in an object.

Let's take a look at an example:

```
let spaceship = {
    crew: {
    captain: {
        name: 'Lily',
        degree: 'Computer Engineering',
        cheerTeam() { console.log('You got this!') }
        },
    'chief officer': {
        name: 'Dan',
        degree: 'Aerospace Engineering',
        agree() { console.log('I agree, captain!') }
        },
    medic: {
        name: 'Clementine',
        degree: 'Physics',
        announce() { console.log(`Jets on!`) } },
    translator: {
        name: 'Shauna',
        degree: 'Conservation Science',
        powerFuel() { console.log('The tank is full!') }
        }
    }
};
```

Using `for...in`, iterate through the `spaceship.crew` object in the code editor and `console.log()` a list of crew roles and names in the following format: `'[crew member's role]: [crew member's name]'`, e.g.,`'chief officer: Dan'`.

```
for (let variableName in outerObject.innerObject) {
  console.log(`${variableName}:
${outerObject.innerObject[variableName].propertyName}`)
};
```

```
for (let crewMember in spaceship.crew) {
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`);
}
```

Using `for...in`, iterate through the `spaceship.crew` object in the code editor and `console.log()` a list of crew names and degrees in the following format: `'[crew member's name]: [crew member's degree]'`, i.e.,`'Lily: Computer Engineering'`.

```
for (let variableName in outerObject.innerObject) {
  console.log(`${outerObject.innerObject[variableName].propertyName}:
${outerObject.innerObject[variableName].differentPropertyName}`)
};
```

```
for (let crewMember in spaceship.crew) {
  console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember].degree}`);
}
```

Remember how we used to format out string in JavaScript?

```
let captain = 'Jean-luc'
console.log('Paging ' + captain + ' to report to the bridge.')
console.log(`Paging ${captain} to report to the bridge.`)
```

Why can't I use crewMember.name?

When using a `for..in` loop, the iterator is a type string (you can verify this with `typeof`) so they do not have the `name` property. For example as it iterates through the captain, `crewMember = 'captain'`.
Since this is a string on its own, you can log it as such. That is why `${crewMember} works correctly. However when you then want to use it to reference, the format doesn't work using dot notation.

```
object.propertyName
```

Dot notation requires the propertyName not as a string. To use it as a string, we can utilize it in associative array notation (with square brackets). `object['propertyName']`.
So in our lesson, this is why you need to format it as `${spaceship.crew[crewMember].name}` This way it checks spaceship and then in crew finds the crewmember and returns the name property of that crewmember.

The 'this' Keyword

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet();
// Output will be "ReferenceError: dietType is not defined"
```

That's strange, why is `dietType` not defined even though it's a property of `goat`? That's because inside the scope of the `.diet()` method, we don't automatically have access to other properties of the `goat` object.

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(this.dietType);
  }
};

goat.diet();
// Output: herbivore
```

The `this` keyword references the *calling object* which provides access to the calling object's properties. In the example above, the calling object is `goat` and by using `this` we're accessing the `goat` object itself, and then the `dietType` property of `goat` by using property dot notation.

Arrow Functions and 'this' Keyword

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet: () => {
    console.log(this.dietType);
  }
};

goat.diet(); // Prints undefined
```

In the comment, you can see that `goat.diet()` would log `undefined`. So what happened? Notice that the `.diet()` method is defined using an arrow function.

Arrow functions inherently *bind*, or tie, an already defined `this` value to the function itself that is NOT the calling object. In the code snippet above, the value of `this` is the *global object*, or an object that exists in the global scope, which doesn't have a `dietType` property and therefore returns `undefined`. Don't use arrow funcs with `this` ...

Privacy (Information Hiding)

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties. When discussing *privacy* in objects, we define it as the idea that only certain properties should be mutable or able to change in value.

Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore `_` before the name of a property to mean that the property should not be altered. Here's an example of using `_` to prepend a property.

```
Const bankAccount = {
  _amount: 1000
}
```

Notice that a funky string is printed to the console! This is known as a side-effect of type-coercion. No need to worry about what this means for now, but it's important to understand that you can cause unwanted side-effects when mutating objects and their properties.

```
const robot = {
  _energyLevel: 'high',
  recharge(){
    this._energyLevel += 30;
    console.log(`Recharged! Energy is currently at ${this._energyLevel}%.`)
  }
};

robot.recharge();
```

Output:

```
Recharged! Energy is currently at high30%.
```

Getters

Getters are methods that get and return the internal properties of an object. But they can do more than just retrieve the value of a property! Let's take a look at a getter method:

```
const person = {
  _firstName: 'John',
  _lastName: 'Doe',
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName} ${this._lastName}`;
    } else {
      return 'Missing a first name or a last name.';
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

Notice that in the getter method above:

- We use the `get` keyword followed by a function.
- We use an `if...else` conditional to check if both `_firstName` and `_lastName` exist (by making sure they both return truthy values) and then return a different value depending on the result.
- We can access the calling object's internal properties using `this`. In `fullName`, we're accessing both `this._firstName` and `this._lastName`.
- In the last line we call `fullName` on `person`. In general, getter methods do not need to be called with a set of parentheses. Syntactically, it looks like we're accessing a property.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an **infinite call stack error**. One workaround is to add an underscore before the property name like we did in the example above.

All getters are methods, but not all methods are getters. Only those specially declared with `get` are getters, and they always return the value of the variable to which they are associated. Their purpose is pretty well defined.

Setters

Along with getter methods, we can also create *setter* methods which reassign values of existing properties within an object. Let's see an example of a setter method:

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a number to age');
    }
  }
};
```

Notice that in the example above:

- We can perform a check for what value is being assigned to `this._age`.
- When we use the setter method, only values that are numbers will reassign `this._age`
- There are different outputs depending on what values are used to reassign `this._age`.

Then to use the setter method:

```
person.age = 40;
console.log(person._age); // Logs: 40
person.age = '40'; // Logs: You must assign a number to age
```

Factory Functions

So far we've been creating objects individually, but there are times where we want to create many instances of an object quickly. Here's where *factory functions* come in. A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

```
const monsterFactory = (name, age, energySource,
catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

In the `monsterFactory` function above, it has four parameters and returns an object that has the properties: `name`, `age`, `energySource`, and `scare()`. To make an object that represents a specific monster like a ghost, we can call `monsterFactory` with the necessary arguments and assign the return value to a variable:

```
const ghost = monsterFactory('Ghouly', 251, 'ectoplasm',
'BOO!');
ghost.scare(); // 'BOO!'
```

Now we have a `ghost` object as a result of calling `monsterFactory()` with the needed arguments.

Question

How would you know when to use factory functions?

Answer

Since Factory Functions are meant to create alike objects in a way that makes it easier to produce in mass quantities, a possible example when you would think on using a factory function instead of making a regular object literal would be a social network where let's say each profile is an object, if we wanted a very minimalistic user with only name, image, friends and posts we could dynamically create them with a factory function where we could also have methods to add friends and create posts:

```
const makeUser = (id, name, image) => {
  return({
    id,
    name,
    image,
    friends: [],
    posts: [],
    addFriend (friend) {
       this.friends.push(friend);
    },
    createPost (post) {
      this.posts.push(post);
    }
  })
}
```

Now every time a user wants to sign up in our very simple social network app when they click sign up, this function will be triggered and they will become a new user object, where if they decide to add a friend the `.addFriend()` method will trigger with the friend to be added to the friends array.

Property Value Shorthand

ES6 introduced some new shortcuts for assigning properties to variables known as *destructuring*.

In the previous exercise, we created a factory function that helped us create objects. We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it. To remind ourselves, here's a truncated version of the factory function:

```
const monsterFactory = (name, age) => {
  return {
    name: name,
    age: age
  }
};
```

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a destructuring technique, called *property value shorthand*, to save ourselves some keystrokes. The example below works exactly like the example above:

```
const monsterFactory = (name, age) => {
  return {
    name,
    age
  }
};
```

*Note that Internet Explorer 11 is not compatible with this concept.

Destructured Assignment

We often want to extract key-value pairs from objects and save them as variables. Take for example the following object:

```
const vampire = {
  name: 'Dracula',
  residence: 'Transylvania',
  preferences: {
    day: 'stay inside',
    night: 'satisfy appetite'
  }
};
```

If we wanted to extract the `residence` property as a variable, we could use the following code:

```
const residence = vampire.residence;
console.log(residence); // Prints 'Transylvania'
```

However, we can also take advantage of a destructuring technique called *destructured assignment* to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object's key that is wrapped in curly braces `{ }` and assign to it the object. Take a look at the example below:

```
const { residence } = vampire;
console.log(residence); // Prints 'Transylvania'
```

Look back at the `vampire` object's properties in the first code example. Then, in the example above, we declare a new variable `residence` that extracts the value of the `residence` property of `vampire`. When we log the value of `residence` to the console, `'Transylvania'` is printed.

We can even use destructured assignment to grab nested properties of an object:

```
const { day } = vampire.preferences;
console.log(day); // Prints 'stay inside'
```

Built-in Object Methods

- **Object.keys()**

The `Object.keys()`method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

```
const object1 = {

  a: 'somestring',

  b: 42,

  c: false

};

console.log(Object.keys(object1));

// expected output: Array ["a", "b", "c"]
```

- **Object.entries()**

```
const object1 = {

  a: 'somestring',

  b: 42

};

for (const [key, value] of Object.entries(object1)) {

  console.log(`${key}: ${value}`);

}

// expected output:

// "a: somestring"

// "b: 42"
```

- **Object.assign()**

The `Object.assign()` method copies all enumerable own properties from one or more *source objects* to a *target object*. It returns the modified target object.

Properties in the target object are overwritten by properties in the sources if they have the same key. Later sources' properties overwrite earlier ones.

```
const target = { a: 1, b: 2 };

const source = { b: 4, c: 5 };


const returnedTarget = Object.assign(target, source);


console.log(target);

// expected output: Object { a: 1, b: 4, c: 5 }


console.log(returnedTarget);

// expected output: Object { a: 1, b: 4, c: 5 }
```

Use of this function:

What if we want another object that has the properties of `robot` but with a few additional properties. `Object.assign()` sounds like a great method to use, but like the previous examples we should check [Object.assign() documentation at MDN]().

```
const newRobot = Object.assign({
  laserBlaster: true,
  voiceRecognition: true
}, robot);
```