Ecma Concepts & Beyond Cheat Sheet

Argument Example

The arguments object is an **array-like object** *(in that the structure of the object is similar to that of an array; however it should not be considered an array as it has all the functionality of an object)* that stores all of the arguments that you passed to a function and is proprietary to that function in particular.

If you were to pass 3 arguments to a function, say `storeNames()`, those 3 arguments would be stored inside an object called **arguments** and it would look like this when we pass the arguments `storeNames("Mulder", "Scully", "Alex Krycek")` to our function:

- First, we declare a function and make it return the arguments object.

```
function storeNames() { return arguments; }
```

- Then, when we execute that function with **n arguments**, 3 in this case, it will return the object to us and it will **look like** an array. We can convert it to an array, but more on that later...

```
// If we execute the following line in the console:
storeNames("Mulder", "Scully", "Alex Kryceck");
// The output will be { '0': 'Mulder', '1': 'Scully', '2': 'Alex Kryceck' }
```

Treat it as an array

You can invoke arguments by using `arguments[n]` (where *n* is the index of the argument in the array-like object). But if you want to use it as an array for iteration purposes or applying array methods to it, you need to *convert it to an array* by declaring a variable and using the Array.prototype.slice.call method (because *arguments* is not an array):

```
var args = Array.prototype.slice.call(arguments);

// or the es6 way:
var args = Array.from(arguments)
```

Since **slice()** has two (the parameter **end** is optional) parameters. You can grab a certain portion of the arguments by specifying the beginning and the ending of your portion (using the *slice.call()* method renders these two parameters optional, not just *end*). Check out the following code:

```
function getGrades() {
    var args = Array.prototype.slice.call(arguments, 1, 3);
    return args;
}

// Let's output this!
console.log(getGrades(90, 100, 75, 40, 89, 95));

// OUTPUT SHOULD BE: //
// [100, 75] <- Why? Because it started from index 1 and stopped at index 3
// so, index 3 (40) wasn't taken into consideration.
//
// If we remove the '3' parameter, leaving just (arguments, 1) we'd get
// every argument from index 1: [100, 75, 40, 89, 95].
```

Optimization issues with Array.slice()

There is a little problem: it's not recommended to use slice in the arguments object (optimization reasons)...

Important: You should not slice on arguments because it prevents optimizations in JavaScript engines (V8 for example). Instead, try constructing a new array by iterating through the arguments object.

So, what other method is available to convert *arguments* to an array? I recommend the for-loop (not the for-in loop). You can do it like this:

```
var args = []; // Empty array, at first.
for (var i = 0; i < arguments.length; i++) {
    args.push(arguments[i])
} // Now 'args' is an array that holds your arguments.
```

For more information on the optimization issues:
Optimization Killers: Managing Arguments

ES6 rest parameter as a way to circumvent the arguments object

In ES2015/ES6 it is possible to use the rest parameter (`...`) instead of the arguments object in most places. Say we have the following function (non-ES6):

```
function getIntoAnArgument() {
    var args = arguments.slice();
    args.forEach(function(arg) {
        console.log(arg);
    });
}
```

```
}
```

That function can be replaced in ES6 by:

```
function getIntoAnArgument(...args) {
    args.forEach(arg => console.log(arg));
}
```

Note that we also used an arrow function to shorten the forEach callback!

The arguments object is not available inside the body of an arrow function.

The rest parameter must always come as the last argument in your function definition. `function getIntoAnArgument(arg1, arg2, arg3, ...restOfArgs /*no more arguments allowed here*/) { //function body }`

Arithmetic Operation Example

JavaScript provides the user with five arithmetic operators: `+`, `-`, `*`, `/` and `%`. The operators are for addition, subtraction, multiplication, division and remainder, respectively.

Addition

Syntax

`a + b`

Usage

```
2 + 3          // returns 5
true + 2       // interprets true as 1 and returns 3
false + 5      // interprets false as 0 and returns 5
true + "bar"   // concatenates the boolean value and returns "truebar"
5 + "foo"      // concatenates the string and the number and returns "5foo"
"foo" + "bar"  // concatenates the strings and returns "foobar"
```

Hint: There is a handy increment operator that is a great shortcut when you're adding numbers by 1.

Subtraction

Syntax

`a - b`

Usage

```
2 - 3      // returns -1
3 - 2      // returns 1
false - 5  // interprets false as 0 and returns -5
true + 3   // interprets true as 1 and returns 4
5 + "foo"  // returns NaN (Not a Number)
```

Hint: There is a handy decrement operator that is a great shortcut when you're subtracting numbers by 1.

Multiplication

Syntax

`a * b`

Usage

```
2 * 3               // returns 6
3 * -2              // returns -6
false * 5           // interprets false as 0 and returns 0
true * 3            // interprets true as 1 and returns 3
5 * "foo"           // returns NaN (Not a Number)
Infinity * 0        // returns NaN
Infinity * Infinity  // returns Infinity
```

Division

Syntax

`a / b`

Usage

```
3 / 2               // returns 1.5
3.0 / 2/0           // returns 1.5
3 / 0               // returns Infinity
3.0 / 0.0           // returns Infinity
-3 / 0              // returns -Infinity
false / 5           // interprets false as 0 and returns 0
true / 2            // interprets true a 1 and returns 0.5
5 + "foo"           // returns NaN (Not a Number)
Infinity / Infinity  // returns NaN
```

Remainder

Syntax

`a % b`

Usage

```
3 % 2          // returns 1
true % 5       // interprets true as 1 and returns 1
false % 4      // interprets false as 0 and returns 0
3 % "bar"      // returns NaN
```

Increment

Syntax

`a++ or ++a`

Usage

// Postfix x = 3; // declare a variable y = x++; // y = 4, x = 3

// Prefix var a = 2; b = ++a; // a = 3, b = 3

Decrement

Syntax

`a-- or --a`

Usage

// Postfix x = 3; // declare a variable y = x—; // y = 3, x = 3

// Prefix var a = 2; b = —a; // a = 1, b = 1 *!Important!* As you can see, you **cannot** perform any sort of operations on `Infinity`.

Arrow Function Example

Arrow functions are a new ES6 syntax for writing JavaScript function expressions. The shorter syntax saves time, as well as simplifying the function scope.

What are arrow functions?

An arrow function expression is a more concise syntax for writing function expressions using a “fat arrow” token (`=>`).

The basic syntax

Below is a basic example of an arrow function:

```
// ES5 syntax
var multiply = function(x, y) {
  return x * y;
};

// ES6 arrow function
var multiply = (x, y) => { return x * y; };

// Or even simpler
var multiply = (x, y) => x * y;
```

You no longer need the `function` and `return` keywords, or even the curly brackets.

A simplified `this`

Before arrow functions, new functions defined their own `this` value. To use `this` inside a traditional function expression, we have to write a workaround like so:

```
// ES5 syntax
function Person() {
  // we assign `this` to `self` so we can use it later
  var self = this;
  self.age = 0;

  setInterval(function growUp() {
    // `self` refers to the expected object
    self.age++;
  }, 1000);
}
```

An arrow function doesn't define its own `this` value, it inherits `this` from the enclosing function:

```
// ES6 syntax
function Person(){
  this.age = 0;

  setInterval(() => {
    // `this` now refers to the Person object, brilliant!
    this.age++;
  }, 1000);
}

var p = new Person();
```

Assignment Operators

Assignment Operator Example

Assignment operators, as the name suggests, assign (or re-assign) values to a variable. While there are quite a few variations on the assignment operators, they all build off of the basic assignment operator.

Syntax = **y;DescriptionNecessityxVariableRequired=Assignment operatorRequiredyValue to assign to variableRequired**

Examples

```
let initialVar = 5;   // Variable initialization requires the use of an assignment operator

let newVar = 5;
newVar = 6;   // Variable values can be modified using an assignment operator
```

Variations

The other assignment operators are a shorthand for performing some operation using the variable (indicated by x above) and value (indicated by y above) and then assigning the result to the variable itself.

For example, below is the syntax for the addition assignment operator:

```
x += y;
```

This is the same as applying the addition operator and reassigning the sum to the original variable (that is, x), which can be expressed by the following code:

```
x = x + y;
```

To illustrate this using actual values, here is another example of using the addition assignment operator:

```
let myVar = 5;   // value of myVar: 5
myVar += 7;   // value of myVar: 12 = 5 + 7
```

Boolean Example

Booleans are a primitive datatype commonly used in computer programming languages. By definition, a boolean has two possible values: `true` or `false`.

In Javascript, there is often implicit type coercion to boolean. If for example you have an if statement which checks a certain expression, that expression will be coerced to a boolean:

```
var a = 'a string';
if (a) {
  console.log(a); // logs 'a string'
}
```

There are only a few values that will be coerced to false:

- false (not really coerced, as it already is false)
- null
- undefined
- NaN
- 0
- '' (empty string)

All other values will be coerced to true. When a value is coerced to a boolean, we also call that either 'falsy' or 'truthy'.

One way that type coercion is used is with the use of the or (`||`) and and (`&&`) operators:

```
var a = 'word';
var b = false;
var c = true;
var d = 0
var e = 1
var f = 2
var g = null

console.log(a || b); // 'word'
console.log(c || a); // true
console.log(b || a); // 'word'
console.log(e || f); // 1
console.log(f || e); // 2
console.log(d || g); // null
console.log(g || d); // 0
console.log(a && c); // true
console.log(c && a); // 'word'
```

As you can see, the *or* operator checks the first operand. If this is true or truthy, it returns it immediately (which is why we get 'word' in the first case & true in the second case). If it is not true or truthy, it returns the second operand (which is why we get 'word' in the third case).

With the and operator it works in a similar way, but for 'and' to be true, both operands need to be truthy. So it will always return the second operand if both are true/truthy, otherwise it will return false. That is why in the fourth case we get true and in the last case we get 'word'.

The Boolean Object

There is also a native JavaScript object that wraps around a value. The value passed as the first parameter is converted to a boolean value, if necessary. If the value is omitted, 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false. All other values, including any object or the string “false”, create an object with an initial value of true.

Do not confuse the primitive Boolean values true and false with the true and false values of the Boolean object.

More Details

Any object whose value is not undefined or null, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. If true, this will execute the function. For example, the condition in the following if statement evaluates to true:

```
var x = new Boolean(false);
if (x) {
  // this code is executed
}
```

This behavior does not apply to Boolean primitives. For example, the condition in the following if statement evaluates to false:

```
var x = false;
if (x) {
  // this code is not executed
}
```

Do not use a Boolean object to convert a non-boolean value to a boolean value. Instead, use Boolean as a function to perform this task:

```
var x = Boolean(expression);     // preferred
var x = new Boolean(expression); // don't use
```

If you specify any object, including a Boolean object whose value is false, as the initial value of a Boolean object, the new Boolean object has a value of true.

```
var myFalse = new Boolean(false);   // initial value of false
var g = new Boolean(myFalse);       // initial value of true
var myString = new String('Hello'); // string object
```

```
var s = new Boolean(myString);      // initial value of true
```

Do not use a Boolean object in place of a Boolean primitive.

Callback Functions

This section gives a brief introduction to the concept and usage of callback functions in Javascript.

Functions are Objects

The first thing we need to know is that in Javascript, functions are first-class objects. As such, we can work with them in the same way we work with other objects, like assigning them to variables and passing them as arguments into other functions. This is important, because it's the latter technique that allows us to extend functionality in our applications.

Callback Function Example

A **callback function** is a function that is passed *as an argument* to another function, to be "called back" at a later time.

A function that accepts other functions as arguments is called a **higher-order function**, which contains the logic for *when* the callback function gets executed. It's the combination of these two that allow us to extend our functionality.

To illustrate callbacks, let's start with a simple example:

```
function createQuote(quote, callback){
  var myQuote = "Like I always say, " + quote;
  callback(myQuote); // 2
}

function logQuote(quote){
  console.log(quote);
}

createQuote("eat your vegetables!", logQuote); // 1

// Result in console:
// Like I always say, eat your vegetables!
```

In the above example, `createQuote` is the higher-order function, which accepts two arguments, the second one being the callback. The `logQuote` function is being used to pass in as our callback function. When we execute the `createQuote` function *(1)*, notice that we are *not appending* parentheses to `logQuote` when we pass it in as an argument. This is because we do not want to execute our callback function right away, we simply want to pass the function definition along to the higher-order function so that it can be

executed later.

Also, we need to ensure that if the callback function we pass in expects arguments, we supply those arguments when executing the callback *(2)*. In the above example, that would be the `callback(myQuote);` statement, since we know that `logQuote` expects a quote to be passed in.

Additionally, we can pass in anonymous functions as callbacks. The below call to `createQuote` would have the same result as the above example:

```
createQuote("eat your vegetables!", function(quote){
  console.log(quote);
});
```

Incidentally, you don't *have* to use the word "callback" as the name of your argument. Javascript just needs to know that it's the correct argument name. Based on the above example, the below function will behave in exactly the same manner.

```
function createQuote(quote, functionToCall) {
  var myQuote = "Like I always say, " + quote;
  functionToCall(myQuote);
}
```

Why use Callbacks?

Most of the time we are creating programs and applications that operate in a **synchronous** manner. In other words, some of our operations are started only after the preceding ones have completed.

Often when we request data from other sources, such as an external API, we don't always know *when* our data will be served back. In these instances we want to wait for the response, but we don't always want our entire application grinding to a halt while our data is being fetched. These situations are where callback functions come in handy.

Let's take a look at an example that simulates a request to a server:

```
function serverRequest(query, callback){
  setTimeout(function(){
    var response = query + "full!";
    callback(response);
  },5000);
}

function getResults(results){
  console.log("Response from the server: " + results);
}

serverRequest("The glass is half ", getResults);
```

```
// Result in console after 5 second delay:
// Response from the server: The glass is half full!
```

In the above example, we make a mock request to a server. After 5 seconds elapse, the response is modified and then our callback function `getResults` gets executed. To see this in action, you can copy/paste the above code into your browser's developer tool and execute it.

Also, if you are already familiar with `setTimeout`, then you've been using callback functions all along. The anonymous function argument passed into the above example's `setTimeout` function call is also a callback! So the example's original callback is actually executed by another callback. Be careful not to nest too many callbacks if you can help it, as this can lead to something called "callback hell"! As the name implies, it isn't a joy to deal with.

JavaScript Class Example

JavaScript does not have the concept of classes inherently.

But we could simulate the functionalities of a class by taking advantage of the prototypal nature of JavaScript.

This section assumes that you have a basic understanding of prototypes.

For the sake of clarity, let us assume that we want to create a class which can do the following

```
var p = new Person('James','Bond'); // create a new instance of Person class
    p.log() // Output: 'I am James Bond' // Accessing a function in the class
    // Using setters and getters
    p.profession = 'spy'
    p.profession // output: James bond is a spy
```

Using class keyword

Like in any other programming language, you can now use the `class` keyword to create a class.

This is not supported in older browsers and was introduced in ECMAScript 2015.

`class` is just a syntactic sugar over JavaScript's existing prototype-based inheritance model.

In general, programmers use the following ways to create a class in JavaScript.

Using methods added to prototypes:

Here, all the methods are added to prototype

```
function Person(firstName, lastName) {
    this._firstName = firstName;
    this._lastName = lastName;
}

Person.prototype.log = function() {
    console.log('I am', this._firstName, this._lastName);
}

// This line adds getters and setters for the profession object. Note that in general you could ju
// Since in this example we are trying the mimic the class above, we try to use the getters and se
Object.defineProperty(Person.prototype, 'profession', {
    set: function(val) {
        this._profession = val;
    },
    get: function() {
        console.log(this._firstName, this._lastName, 'is a', this._profession);
    }
})
```

You could also write prototype methods over function `Person` as below:

```
Person.prototype = {
    log: function() {
        console.log('I am ', this._firstName, this._lastName);
    }
    set profession(val) {
        this._profession = val;
    }

    get profession() {
        console.log(this._firstName, this._lastName, 'is a', this._profession);
    }

}
```

Using methods added internally

Here the methods are added internally instead of prototype:

```
function Person(firstName, lastName) {
    this._firstName = firstName;
    this._lastName = lastName;

    this.log = function() {
        console.log('I am ', this._firstName, this._lastName);
    }

    Object.defineProperty(this, 'profession', {
        set: function(val) {
            this._profession = val;
        },
        get: function() {
            console.log(this._firstName, this._lastName, 'is a', this._profession);
        }
    })
}
```

```
}
```

Hiding details in classes with symbols

Most often, some properties and methods have to be hidden to prevent access from outside the function.

With classes, to obtain this functionality, one way to do this is by using symbols. Symbol is a new built-in type of JavaScript, which can be invoked to give a new symbol value. Every Symbol is unique and can be used as a key on object.

So one use case of symbols is that you can add something to an object you might not own, and you might not want to collide with any other keys of object. Therefore, creating a new one and adding it as a property to that object using symbol is the safest. Also, when symbol value is added to an object, no one else will know how to get it.

```
class Person {
    constructor(firstName, lastName) {
        this._firstName = firstName;
        this._lastName = lastName;
    }

    log() {
        console.log('I am', this._firstName, this._lastName);
    }

    // setters
    set profession(val) {
        this._profession = val;
    }
    // getters
    get profession() {
        console.log(this._firstName, this._lastName, 'is a', this._profession);
    }
// With the above code, even though we can access the properties outside the function to change th
// Symbols come to rescue.
let s_firstname  = new Symbol();

class Person {
    constructor(firstName, lastName) {
        this[s_firstName] = firstName;
        this._lastName = lastName;
    }

    log() {
        console.log('I am', this._firstName, this._lastName);
    }

    // setters
    set profession(val) {
        this._profession = val;
    }
    // getters
    get profession() {
        console.log(this[s_firstName], this._lastName, 'is a', this._profession);
    }
```

JavaScript Closure Example

A closure is the combination of a function and the lexical environment (scope) within

which that function was declared. Closures are a fundamental and powerful property of Javascript. This section discusses the 'how' and 'why' about Closures:

Example

```
//we have an outer function named walk and an inner function named fly

function walk (){

  var dist = '1780 feet';

  function fly(){
    console.log('At '+dist);
  }

  return fly;
}

var flyFunc = walk(); //calling walk returns the fly function which is being assigned to flyFunc
//you would expect that once the walk function above is run
//you would think that JavaScript has gotten rid of the 'dist' var

flyFunc(); //Logs out 'At 1780 feet'
//but you still can use the function as above
//this is the power of closures
```

Another Example

```
function by(propName) {
    return function(a, b) {
        return a[propName] - b[propName];
    }
}

const person1 = {name: 'joe', height: 72};
const person2 = {name: 'rob', height: 70};
const person3 = {name: 'nicholas', height: 66};

const arr_ = [person1, person2, person3];

const arr_sorted = arr_.sort(by('height')); // [ { name: 'nicholas', height: 66 }, { name: 'rob',
```

The closure 'remembers' the environment in which it was created. This environment consists of any local variables that were in-scope at the time the closure was created.

```
function outside(num) {
  var rememberedVar = num; // In this example, rememberedVar is the lexical environment that the c
  return function inside() { // This is the function which the closure 'remembers'
    console.log(rememberedVar)
  }
}

var remember1 = outside(7); // remember1 is now a closure which contains rememberedVar = 7 in its
var remember2 = outside(9); // remember2 is now a closure which contains rememberedVar = 9 in its

remember1(); // This now executes the function 'inside' which console.logs(rememberedVar) => 7
remember2(); // This now executes the function 'inside' which console.logs(rememberedVar) => 9
```

Closures are useful because they let you 'remember' data and then let you operate on

that data through returned functions. This allows Javascript to emulate private methods that are found in other programming languages. Private methods are useful for restricting access to code as well as managing your global namespace.

Private variables and methods

Closures can also be used to encapsulate private data/methods. Take a look at this example:

```
const bankAccount = (initialBalance) => {
  const balance = initialBalance;

  return {
    getBalance: function() {
      return balance;
    },
    deposit: function(amount) {
      balance += amount;
      return balance;
    },
  };
};

const account = bankAccount(100);

account.getBalance(); // 100
account.deposit(10); // 110
```

In this example, we won't be able to access `balance` from anywhere outside of the `bankAccount` function, which means we've just created a private variable.

Where's the closure? Well, think about what `bankAccount()` is returning. It actually returns an Object with a bunch of functions inside it, and yet when we call `account.getBalance()`, the function is able to "remember" its initial reference to `balance`.

That is the power of the closure, where a function "remembers" its lexical scope (compile time scope), even when the function is executed outside that lexical scope.

Emulating block-scoped variables

Javascript did not have a concept of block-scoped variables. Meaning that when defining a variable inside a for-loop, for example, this variable was visible from outside the for-loop as well. So how can closures help us solve this problem? Let's take a look.

```
var funcs = [];

for(var i = 0; i < 3; i++){
    funcs[i] = function(){
        console.log('My value is ' + i);  //creating three different functions with different
    }
}

for(var j = 0; j < 3; j++){
    funcs[j]();             // My value is 3
```

```
                                // My value is 3
                                // My value is 3
}
```

Since the variable i does not have block-scope, it's value within all three functions was updated with the loop counter and created malicious values. Closures can help us solve this issue by creating a snapshot of the environment the function was in when it was created, preserving its state.

```
var funcs = [];

var createFunction = function(val){
    return function() {console.log("My value: " + val);};
}

for (var i = 0; i < 3; i++) {
    funcs[i] = createFunction(i);
}
for (var j = 0; j < 3; j++) {
    funcs[j]();                 // My value is 0
                                // My value is 1
                                // My value is 2
}
```

The later versions of Javascript (ES6+) have a new keyword called let which can be used to give the variable a blockscope. There are also many functions (forEach) and entire libraries (lodash.js) that are dedicated to solving such problems as the ones explained above. They can certainly boost your productivity, however it remains extremely important to have knowledge of all these issues when attempting to create something big.

Closures have many special applications that are useful when creating large Javascript programs.

1. Emulating private variables or encapsulation
2. Making Asynchronous server side calls
3. Creating a block-scoped variable.

Emulating private variables

Unlike many other languages, Javascript does not have a mechanism which allows you to create encapsulated instance variables within an object. Having public instance variables can cause a lot of problems when building medium to large programs. However with closures, this problem can be mitigated.

Much like in the previous example, you can build functions which return object literals with methods that have access to the object's local variables without exposing them. Thus, making them effectively private.

Closures can also help you manage your global namespace to avoid collisions with globally shared data. Usually, all global variables are shared between all scripts in your project, which will definitely give you a lot of trouble when building medium to large programs.

That is why library and module authors use closures to hide an entire module's methods and data. This is called the module pattern, it uses an immediately invoked function expression which exports only certain functionality to the outside world, significantly reducing the amount of global references.

Here's a short sample of a module skeleton.

```
var myModule = (function() {
    let privateVariable = 'I am a private variable';

    let method1 = function(){ console.log('I am method 1'); };
    let method2 = function(){ console.log('I am method 2, ', privateVariable); };

    return {
        method1: method1,
        method2: method2
    }
}());

myModule.method1(); // I am method 1
myModule.method2(); // I am method 2, I am a private variable
```

Closures are useful for capturing new instances of private variables contained in the 'remembered' environment, and those variables can only be accessed through the returned function or methods.

Block scoping

Let

```
function fn () {
  let x = 0
  if (true) {
    let x = 1 // only inside this `if`
  }
}
```

Const

```
const a = 1
```

`let` is the new `var`. Constants work just like `let`, but can't be reassigned. See: Let and const

New methods

New string methods

```
"hello".repeat(3)
"hello".includes("ll")
"hello".startsWith("he")
"\u1E9B\u0323".normalize("NFC")
```

See: New methods

Exponent operator

```
const byte = 2 ** 8
// Same as: Math.pow(2, 8)
```

Backtick strings

Interpolation

```
const message = `Hello ${name}`
```

Multiline strings

```
const str = `
hello
world
`
```

Templates and multiline strings. See: Template strings

Binary and octal literals

```
let bin = 0b1010010
let oct = 0o755
```

See: Binary and octal literals

Classes

```
class Circle extends Shape {
```

Constructor

```
  constructor (radius) {
    this.radius = radius
  }
```

Methods

```
  getArea () {
    return Math.PI * 2 * this.radius
  }
```

Calling superclass methods

```
  expand (n) {
    return super.expand(n) * Math.PI
  }
```

Static methods

```
  static createFromDiameter(diameter) {
    return new Circle(diameter / 2)
  }
}
```

Syntactic sugar for prototypes. See: Classes

Using const and let

```
1  const a = 1
2  let b = 'foo'
3
4  // Not allowed!
5  // a = 2
6
7  // Ok!
8  b = 'bar'
9
10 if (true) {
11   const a = 3
12 }
```

No Errors Show Details

Output compiled with Babel

```
1  var a = 1;
2  var b = 'foo';
3
4  // Not allowed!
5  // a = 2
6
7  // Ok!
8  b = 'bar';
9
10 if (true) {
11   var _a = 3;
12 }
```

Using class

```
  static multiply(value1, value2) {
    return value1 * value2
  }

  sum() {
    return this.value1 + this.value2
  }
}

const calc = new Calculator(2, 3)

console.log(calc.sum())
console.log(Calculator.multiply(2, 3))
```

No Errors Show Details

Console output

```
5
6
```

`class` gives us simple inheritance with the keyword `extends`. Classes that inherit from a parent have access to respective parent functions via `super`.

Inheritance

```

class SquareCalculator {
  constructor(value) {
    this.value = value
  }

  calculate() {
    return this.value * this.value
  }
}

class CubeCalculator extends SquareCalculator {
  calculate() {
```

No Errors Show Details

Console output

```
27
```

For full details on the `class` syntax, see the MDN reference.

Making promises

```
new Promise((resolve, reject) => {
  if (ok) { resolve(result) }
  else { reject(error) }
})
```

For asynchronous programming. See: Promises

Using promises

```
promise
  .then((result) => { ··· })
  .catch((error) => { ··· })
```

Async-await

```
async function run () {
  const user = await getUser()
  const tweets = await getTweets(user)
  return [user, tweets]
}
```

async functions are another way of using functions.

See: async function

Promise functions

```
Promise.all(···)
Promise.race(···)
Promise.reject(···)
Promise.resolve(···)
```

Destructuring assignment

Arrays

```
const [first, last] = ['Nikola', 'Tesla']
```

Objects

```
let {title, author} = {
  title: 'The Silkworm',
  author: 'R. Galbraith'
}
```

Supports for matching arrays and objects. See: Destructuring

Loops

```
for (let {title, artist} of songs) {
  ···
}
```

The assignment expressions work in loops, too.

Default values

```
const scores = [22, 33]
const [math = 50, sci = 50, arts = 50] = scores

// Result:
// math === 22, sci === 33, arts === 50
```

Default values can be assigned while destructuring arrays or objects.

Default values

```
function greet({ name = 'Rauno' } = {}) {
  console.log(`Hi ${name}!`);
}

greet() // Hi Rauno!
greet({ name: 'Larry' }) // Hi Larry!
```

Function arguments

```
function greet({ name, greeting }) {
  console.log(`${greeting}, ${name}!`)
}

greet({ name: 'Larry', greeting: 'Ahoy' })
```

Destructuring of objects and arrays can be also be done in function arguments.

Reassigning keys

```
function printCoordinates({ left: x, top: y }) {
  console.log(`x: ${x}, y: ${y}`)
}

printCoordinates({ left: 25, top: 90 })
```

This example assigns `x` to the value of the `left` key.

Object spread

with Object spread

```
const options = {
  ...defaults,
  visible: true
}
```

without Object spread

```
const options = Object.assign(
  {}, defaults,
  { visible: true })
```

The Object spread operator lets you build new objects from other objects.

See: Object spread

Array spread

with Array spread

```
const users = [
  ...admins,
  ...editors,
  'rstacruz'
]
```

without Array spread

```
const users = admins
  .concat(editors)
  .concat([ 'rstacruz' ])
```

The spread operator lets you build new arrays in the same way.

See: Spread operator

Function arguments

Default arguments

```
function greet (name = 'Jerry') {
  return `Hello ${name}`
}
```

Rest arguments

```
function fn(x, ...y) {
  // y is an Array
  return x * y.length
}
```

Spread

```
fn(...[1, 2, 3])
// same as fn(1, 2, 3)
```

Default, rest, spread. See: Function arguments

Fat arrows

Fat arrows

```
setTimeout(() => {
  ···
})
```

With arguments

```
readFile('text.txt', (err, data) => {
  ...
})
```

Implicit return

```
numbers.map(n => n * 2)
// No curly braces = implicit return
// Same as: numbers.map(function (n) { return n * 2 })
```

Like functions but with `this` preserved. See: Fat arrows

Fat arrow functions

```
const foo = () => 'bar'

const baz = (x) => {
  return x + 1
}

const squareSum = (...args) => {
  const squared = args.map(x => Math.pow(x, 2))
  return squared.reduce((prev, curr) => prev + curr)
}

this.items.map(x => this.doSomethingWith(x))
```

No Errors Show Details

Output compiled with Babel

```
var _this = this;

var foo = function foo() {
  return 'bar';
};

var baz = function baz(x) {
  return x + 1;
};

var squareSum = function squareSum() {
  for (var _len = arguments.length, args = Array(_len), _key =
    args[_key] = arguments[_key];
  }

```

Shorthand syntax

```
module.exports = { hello, bye }
// Same as: module.exports = { hello: hello, bye: bye }
```

See: Object literal enhancements

Getters and setters

```
const App = {
  get closed () {
    return this.status === 'closed'
  },
  set closed (value) {
    this.status = value ? 'closed' : 'open'
  }
}
```

See: Object literal enhancements

Methods

```
const App = {
  start () {
    console.log('running')
  }
}
// Same as: App = { start: function () {···} }
```

See: Object literal enhancements

Computed property names

```
let event = 'click'
let handlers = {
  [`on${event}`]: true
}
// Same as: handlers = { 'onclick': true }
```

See: Object literal enhancements

Imports

```
import 'helpers'
// aka: require('···')

import Express from 'express'
// aka: const Express = require('···').default || require('···')

import { indent } from 'helpers'
// aka: const indent = require('···').indent

import * as Helpers from 'helpers'
// aka: const Helpers = require('···')

import { indentSpaces as indent } from 'helpers'
// aka: const indent = require('···').indentSpaces
```

`import` is the new `require()`. See: Module imports

Exports

```
export default function () { ··· }
// aka: module.exports.default = ···

export function mymethod () { ··· }
// aka: module.exports.mymethod = ···

export const pi = 3.14159
// aka: module.exports.pi = ···
```

`export` is the new `module.exports`. See: Module exports

Generators

```
function* idMaker () {
  let id = 0
  while (true) { yield id++ }
}
```

```
let gen = idMaker()
gen.next().value  // → 0
gen.next().value  // → 1
gen.next().value  // → 2
```

It's complicated. See: Generators

For..of iteration

```
for (let i of iterable) {
  ...
}
```

For iterating through generators and arrays. See: For..of iteration