# JavaScript Standards

## Syntax & Basics

## Coming Soon...

## Coding Guidelines

I. **Variable declaration:**

Bad:

```
const yyyymmdstr = moment().format("YYYY/MM/DD");
```

Good:

```
const currentDate = moment().format("YYYY/MM/DD");
```

II. **Same vocabulary for same type of variable:**

Bad:

```
getUserInfo();
getClientData();
getCustomerRecord();
```

Good:

```
getUser();
```

III. **Naming the variable and making the code meaningful:** Tools like buddy.js and ESLint can help identify unnamed constants.

Bad:

```
// What the heck is 86400000 for?
setTimeout(blastOff, 86400000);
```

Good:

```
// Declare them as capitalized named constants.
const MILLISECONDS_IN_A_DAY = 86400000;
setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

IV. **Explanatory Variables:**

Bad:

```
const address = "One Infinite Loop, Cupertino 95014";
const cityZipCodeRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
saveCityZipCode(
  address.match(cityZipCodeRegex)[1],
  address.match(cityZipCodeRegex)[2]
);
```

Good:

```
const address = "One Infinite Loop, Cupertino 95014";
const cityZipCodeRegex = /^[^,\\]+[,\\\s]+(.+?)\s*(\d{5})?$/;
```

```
const [, city, zipCode] = address.match(cityZipCodeRegex) || [];
saveCityZipCode(city, zipCode);
```

V. **Avoid Mental Mapping:** Explicit is better than implicit

Bad:

```
const locations = ["Austin", "New York", "San Francisco"];
locations.forEach(l => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // Wait, what is `l` for again?
  dispatch(l);
});
```

Good:

```
const locations = ["Austin", "New York", "San Francisco"];
locations.forEach(location => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  dispatch(location);
});
```

VI. **Not needed context:**

Bad:

```
const Bike = {
  bikeMake: "Honda",
  bikeModel: "Kawasaki",
  bikeColor: "Bajaj"
};

function paintBike(bike) {
  bike.bikeColor = "Red";
}
```

Good:

```
const Bike = {
  make: "Honda",
  model: "Kawasaki",
  color: "Bajaj"
};
function paintBike(bike) {
  bike.color = "Red";
}
```

VII. **Use default arguments:**

Bad:

```
function createPodcast(name) {
  const podcastName = name || "Today & Tomorrow";
```

```
  // ...
}
```

Good:

```
function createPodcast(name = "Today & Tomorrow") {
  // ...
}
```

**VIII. Function arguments (2 or fewer ideally):**

Bad:

```
function createSubject(to, subject, body, cancellable) {
  // …
}
```

Good:

```
function createSubject({ to, subject, body, cancellable }) {
  // ...
}
createSubject({
  to: "AddressingTo",
  subject: "Inviting for event",
  body: "Welcome this is body",
  cancellable: true
});
```

**IX. Functions should do one thing:**

Bad:

```
function emailCustomers(customers) {
  customers.forEach(customer => {
    const customerRecord = database.lookup(customer);
    if (customerRecord.isActive()) {
      email(customer);
    }
  });
}
```

Good:

```
function emailActiveCustomers(customers) {
  customers.filter(isActiveCustomer).forEach(email);
}

function isActiveCustomer(customer) {
  const customerRecord = database.lookup(customer);
  return customerRecord.isActive();
}
```

**X. Function names should say what they do:**

Bad:

```
function addToDate(date, month) {
```

```
  // ...
}
const date = new Date();
// It's hard to tell from the function name what is added
addToDate(date, 1);
```

Good:

```
function addMonthToDate(month, date) {
  // ...
}
const date = new Date();
addMonthToDate(1, date);
```

XI. **Functions should only be one level of abstraction:** When you have more than one level of abstraction your function is usually doing too much. Splitting up functions leads to reusability and easier testing.

Bad:

```
function parseBetterJSAlternative(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(" ");
  const tokens = [];
  REGEXES.forEach(REGEX => {
    statements.forEach(statement => {
      // ...
    });
  });

  const ast = [];
  tokens.forEach(token => {
    // lex...
  });

  ast.forEach(node => {
    // parse...
  });
}
```

Good:

```
function parseBetterJSAlternative(code) {
  const tokens = tokenize(code);
  const syntaxTree = parse(tokens);
  syntaxTree.forEach(node => {
    // parse...
  });
}
```

```
function tokenize(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(" ");
  const tokens = [];
  REGEXES.forEach(REGEX => {
    statements.forEach(statement => {
      tokens.push(/* ... */);
    });
  });

  return tokens;
}

function parse(tokens) {
  const syntaxTree = [];
  tokens.forEach(token => {
    syntaxTree.push(/* ... */);
  });

  return syntaxTree;
}
```

XII. **Remove duplicate code:** Do your absolute best to avoid duplicate code. Duplicate code is bad because it means that there's more than one place to alter something if you need to change some logic.
Imagine if you run a restaurant and you keep track of your inventory: all your tomatoes, onions, garlic, spices, etc. If you have multiple lists that you keep this on, then all have to be updated when you serve a dish with tomatoes in them. If you only have one list, there's only one place to update!
Oftentimes you have duplicate code because you have two or more slightly different things, that share a lot in common, but their differences force you to have two or more separate functions that do much of the same things. Removing duplicate code means creating an abstraction that can handle this set of different things with just one function/module/class.

Getting the abstraction right is critical, that's why you should follow the SOLID principles laid out in the Classes section. Bad abstractions can be worse than duplicate code, so be careful! Having said this, if you can make a good abstraction, do it! Don't repeat yourself, otherwise you'll find yourself updating multiple places anytime you want to change one thing.

Bad:

```
function showDeveloperList(developers) {
  developers.forEach(developer => {
    const expectedSalary = developer.calculateExpectedSalary();
    const experience = developer.getExperience();
    const githubLink = developer.getGithubLink();
    const data = {
```

```
      expectedSalary,
      experience,
      githubLink
    };

    render(data);
  });
}

function showManagerList(managers) {
  managers.forEach(manager => {
    const expectedSalary = manager.calculateExpectedSalary();
    const experience = manager.getExperience();
    const portfolio = manager.getMBAProjects();
    const data = {
      expectedSalary,
      experience,
      portfolio
    };

    render(data);
  });
}
```

Good:

```
function showEmployeeList(employees) {
  employees.forEach(employee => {
    const expectedSalary = employee.calculateExpectedSalary();
    const experience = employee.getExperience();

    const data = {
      expectedSalary,
      experience
    };
    switch (employee.type) {
      case "manager":
        data.portfolio = employee.getMBAProjects();
        break;
      case "developer":
        data.githubLink = employee.getGithubLink();
        break;
    }
    render(data);
  });
}
```

**XIII. Set default objects with Object.assign:**

Bad:

```
const menuConfig = {
```

```
  title: null,
  body: "Bar",
  buttonText: null,
  cancellable: true
};
function createMenu(config) {
  config.title = config.title || "Foo";
  config.body = config.body || "Bar";
  config.buttonText = config.buttonText || "Baz";
  config.cancellable =
    config.cancellable !== undefined ? config.cancellable : true;
}
createMenu(menuConfig);
```

Good:

```
const menuConfig = {
  title: "Order",
  // User did not include 'body' key
  buttonText: "Send",
  cancellable: true
};
function createMenu(config) {
  config = Object.assign(
    {
      title: "Foo",
      body: "Bar",
      buttonText: "Baz",
      cancellable: true
    },
    config
  );

  // config now equals: {title: "Order", body: "Bar", buttonText: "Send",
cancellable: true}
  // ...
}
createMenu(menuConfig);
```

XIV. **Don't use flags as function parameters:** Flags tell your user that this function does more than one thing. Functions should do one thing. Split out your functions if they are following different code paths based on a boolean.

Bad:

```
function createFile(name, temp) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
```

```
}
```

Good:

```
function createFile(name) {
  fs.create(name);
}

function createTempFile(name) {
  createFile(`./temp/${name}`);
}
```

XV. **Avoid Side Effects (part 2):** In JavaScript, primitives are passed by value and objects/arrays are passed by reference. In the case of objects and arrays, if your function makes a change in a shopping cart array, for example, by adding an item to purchase, then any other function that uses that cart array will be affected by this addition. That may be great, however it can be bad too. Let's imagine a bad situation:
The user clicks the "Purchase", button which calls a purchase function that spawns a network request and sends the cartarray to the server. Because of a bad network connection, the purchase function has to keep retrying the request. Now, what if in the meantime the user accidentally clicks "Add to Cart" button on an item they don't actually want before the network request begins? If that happens and the network request begins, then that purchase function will send the accidentally added item because it has a reference to a shopping cart array that the addItemToCart function modified by adding an unwanted item.
A great solution would be for the addItemToCart to always clone the cart, edit it, and return the clone. This ensures that no other functions that are holding onto a reference of the shopping cart will be affected by any changes.
Two caveats to mention to this approach:
There might be cases where you actually want to modify the input object, but when you adopt this programming practice you will find that those cases are pretty rare. Most things can be refactored to have no side effects!
Cloning big objects can be very expensive in terms of performance. Luckily, this isn't a big issue in practice because there are great libraries that allow this kind of programming approach to be fast and not as memory intensive as it would be for you to manually clone objects and arrays.
Bad:

```
const addItemToCart = (cart, item) => {
  cart.push({ item, date: Date.now() });
};
```

Good:

```
const addItemToCart = (cart, item) => {
  return [...cart, { item, date: Date.now() }];
};
```

XVI. **Don't write to global functions:** Polluting globals is a bad practice in JavaScript because you could clash with another library and the user of your API would be none-the-wiser until they get an exception in production. Let's think about an example: what if you wanted to extend JavaScript's native Array method to have a `diff` method that could show the difference between two arrays? You could write your new function to the `Array.prototype`, but it could clash with another library that tried to do the same thing. What if that other library was just using `diff` to find the difference between the first and last elements of an array? This is why it would be much better to just use ES2015/ES6 classes and simply extend the `Array` global.

Bad:

```
Array.prototype.diff = function diff(comparisonArray) {
  const hash = new Set(comparisonArray);
  return this.filter(elem => !hash.has(elem));
};
```

Good:

```
class SuperArray extends Array {
  diff(comparisonArray) {
    const hash = new Set(comparisonArray);
    return this.filter(elem => !hash.has(elem));
  }
}
```

XVII. **Favor functional programming over imperative programming:** JavaScript isn't a functional language in the way that Haskell is, but it has a functional flavor to it. Functional languages can be cleaner and easier to test. Favor this style of programming when you can.

Bad:

```
const programmerOutput = [
  {
    name: "Uncle Bobby",
    linesOfCode: 500
  },
  {
    name: "Suzie Q",
    linesOfCode: 1500
  },
  {
    name: "Jimmy Gosling",
    linesOfCode: 150
  },
  {
    name: "Gracie Hopper",
    linesOfCode: 1000
  }
];

let totalOutput = 0;

for (let i = 0; i < programmerOutput.length; i++) {
  totalOutput += programmerOutput[i].linesOfCode;
}
```

Good:

```
const programmerOutput = [
  {
    name: "Uncle Bobby",
    linesOfCode: 500
  },
  {
```

```
    name: "Suzie Q",
    linesOfCode: 1500
  },
  {
    name: "Jimmy Gosling",
    linesOfCode: 150
  },
  {
    name: "Gracie Hopper",
    linesOfCode: 1000
  }
];

const totalOutput = programmerOutput.reduce(
  (totalLines, output) => totalLines + output.linesOfCode,
  0
);
```

**XVIII. Encapsulate conditionals:**

Bad:

```
if (fsm.state === "fetching" && isEmpty(listNode)) {
  // ...
}
```

Good:

```
function shouldShowSpinner(fsm, listNode) {
  return fsm.state === "fetching" && isEmpty(listNode);
}

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {
  // ...
}
```

**XIX. Avoid negative conditionals**

Bad:

```
function isDOMNodeNotPresent(node) {
  // ...
}

if (!isDOMNodeNotPresent(node)) {
  // …
}
```

Good:

```
function isDOMNodePresent(node) {
  // ...
}

if (isDOMNodePresent(node)) {
```

```
  // …
}
```

**XX.** **Avoid conditionals:** This seems like an impossible task. Upon first hearing this, most people say, "how am I supposed to do anything without an `if` statement?" The answer is that you can use polymorphism to achieve the same task in many cases. The second question is usually, "well that's great but why would I want to do that?" The answer is a previous clean code concept we learned: a function should only do one thing. When you have classes and functions that have `if` statements, you are telling your user that your function does more than one thing. Remember, just do one thing.

Bad:

```
class Airplane {
  // ...
  getCruisingAltitude() {
    switch (this.type) {
      case "777":
        return this.getMaxAltitude() - this.getPassengerCount();
      case "Air Force One":
        return this.getMaxAltitude();
      case "Cessna":
        return this.getMaxAltitude() - this.getFuelExpenditure();
    }
  }
}
```

Good:

```
class Airplane {
  // ...
}

class Boeing777 extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getPassengerCount();
  }
}

class AirForceOne extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude();
  }
}

class Cessna extends Airplane {
  // ...
  getCruisingAltitude() {
    return this.getMaxAltitude() - this.getFuelExpenditure();
  }
}
```

**XXI.** **Avoid type-checking (part 1):** JavaScript is untyped, which means your functions can take any type of argument. Sometimes you are bitten by this freedom and it becomes tempting to do type-checking in your functions. There are many ways to avoid having to do this. The first thing to consider is consistent APIs.

Bad:

```
function travelToTexas(vehicle) {
  if (vehicle instanceof Bicycle) {
    vehicle.pedal(this.currentLocation, new Location("texas"));
  } else if (vehicle instanceof Car) {
    vehicle.drive(this.currentLocation, new Location("texas"));
  }
}
```

Good:

```
function travelToTexas(vehicle) {
  vehicle.move(this.currentLocation, new Location("texas"));
}
```

**XXII.** **Avoid type-checking (part 2):** If you are working with basic primitive values like strings and integers, and you can't use polymorphism but you still feel the need to type-check, you should consider using TypeScript. It is an excellent alternative to normal JavaScript, as it provides you with static typing on top of standard JavaScript syntax. The problem with manually type-checking normal JavaScript is that doing it well requires so much extra verbiage that the faux "type-safety" you get doesn't make up for the lost readability. Keep your JavaScript clean, write good tests, and have good code reviews. Otherwise, do all of that but with TypeScript (which, like I said, is a great alternative!).

Bad:

```
function combine(val1, val2) {
  if (
    (typeof val1 === "number" && typeof val2 === "number") ||
    (typeof val1 === "string" && typeof val2 === "string")
  ) {
    return val1 + val2;
  }

  throw new Error("Must be of type String or Number");
}
```

Good:

```
function combine(val1, val2) {
  return val1 + val2;
}
```

**XXIII.** **Don't over-optimize:** Modern browsers do a lot of optimization under-the-hood at runtime. A lot of times, if you are optimizing then you are just wasting your time. There are good resources for seeing where optimization is lacking. Target those in the meantime, until they are fixed if they can be.

Bad:

```
// On old browsers, each iteration with uncached `list.length` would be costly
// because of `list.length` recomputation. In modern browsers, this is optimized.
for (let i = 0, len = list.length; i < len; i++) {
  // ...
```

```
}
```

Good:

```
for (let i = 0; i < list.length; i++) {
  // ...
}
```

XXIV. **Remove dead code:** Dead code is just as bad as duplicate code. There's no reason to keep it in your codebase. If it's not being called, get rid of it! It will still be safe in your version history if you still need it.

Bad:

```
function oldRequestModule(url) {
  // ...
}

function newRequestModule(url) {
  // ...
}

const req = newRequestModule;
inventoryTracker("apples", req, "www.inventory-awesome.io");
```

Good:

```
function newRequestModule(url) {
  // ...
}

const req = newRequestModule;
inventoryTracker("apples", req, "www.inventory-awesome.io");
```

## Objects and Data Structure

XXV. **Use getters and setters:** Using getters and setters to access data on objects could be better than simply looking for a property on an object. "Why?" you might ask. Well, here's an unorganized list of reasons why:

- When you want to do more beyond getting an object property, you don't have to look up and change every accessor in your codebase.
- Makes adding validation simple when doing a set.
- Encapsulates the internal representation.
- Easy to add logging and error handling when getting and setting.
- You can lazy load your object's properties, let's say getting it from a server.

Bad:

```
function makeBankAccount() {
  // ...
```

```
  return {
    balance: 0
    // ...
  };
}

const account = makeBankAccount();
account.balance = 100;
```

Good:

```
function makeBankAccount() {
  // this one is private
  let balance = 0;

  // a "getter", made public via the returned object below
  function getBalance() {
    return balance;
  }

  // a "setter", made public via the returned object below
  function setBalance(amount) {
    // ... validate before updating the balance
    balance = amount;
  }

  return {
    // ...
    getBalance,
    setBalance
  };
}

const account = makeBankAccount();
account.setBalance(100);
```

**XXVI.** **Make objects have private members:** This can be accomplished through closures (for ES5 and below).

Bad:

```
const Employee = function(name) {
  this.name = name;
};

Employee.prototype.getName = function getName() {
  return this.name;
};

const employee = new Employee("John Doe");
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
delete employee.name;
```

```
console.log(`Employee name: ${employee.getName()}`); // Employee name: undefined
```

Good:

```
function makeEmployee(name) {
  return {
    getName() {
      return name;
    }
  };
}

const employee = makeEmployee("John Doe");
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe
```

## Classes

**XXVII.** **Prefer ES2015/ES6 classes over ES5 plain functions:** It's very difficult to get readable class inheritance, construction, and method definitions for classical ES5 classes. If you need inheritance (and be aware that you might not), then prefer ES2015/ES6 classes. However, prefer small functions over classes until you find yourself needing larger and more complex objects.

Bad:

```
const Animal = function(age) {
  if (!(this instanceof Animal)) {
    throw new Error("Instantiate Animal with `new`");
  }

  this.age = age;
};

Animal.prototype.move = function move() {};

const Mammal = function(age, furColor) {
  if (!(this instanceof Mammal)) {
    throw new Error("Instantiate Mammal with `new`");
  }

  Animal.call(this, age);
  this.furColor = furColor;
};

Mammal.prototype = Object.create(Animal.prototype);
Mammal.prototype.constructor = Mammal;
Mammal.prototype.liveBirth = function liveBirth() {};

const Human = function(age, furColor, languageSpoken) {
```

```
  if (!(this instanceof Human)) {
    throw new Error("Instantiate Human with `new`");
  }

  Mammal.call(this, age, furColor);
  this.languageSpoken = languageSpoken;
};

Human.prototype = Object.create(Mammal.prototype);
Human.prototype.constructor = Human;
Human.prototype.speak = function speak() {};
```

Good:

```
class Animal {
  constructor(age) {
    this.age = age;
  }

  move() {
    /* ... */
  }
}

class Mammal extends Animal {
  constructor(age, furColor) {
    super(age);
    this.furColor = furColor;
  }

  liveBirth() {
    /* ... */
  }
}

class Human extends Mammal {
  constructor(age, furColor, languageSpoken) {
    super(age, furColor);
    this.languageSpoken = languageSpoken;
  }

  speak() {
    /* ... */
  }
}
```

**XXVIII.** **Use method chaining:** This pattern is very useful in JavaScript and you see it in many libraries such as jQuery and Lodash. It allows your code to be expressive, and less verbose. For that reason, I say, use method chaining

and take a look at how clean your code will be. In your class functions, simply return `this` at the end of every function, and you can chain further class methods onto it.

Bad:

```
class Car {
  constructor(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
  }

  setMake(make) {
    this.make = make;
  }

  setModel(model) {
    this.model = model;
  }

  setColor(color) {
    this.color = color;
  }

  save() {
    console.log(this.make, this.model, this.color);
  }
}

const car = new Car("Ford", "F-150", "red");
car.setColor("pink");
car.save();
```

Good:

```
class Car {
  constructor(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
  }

  setMake(make) {
    this.make = make;
    // NOTE: Returning this for chaining
    return this;
  }

  setModel(model) {
    this.model = model;
    // NOTE: Returning this for chaining
```

```
    return this;
  }

  setColor(color) {
    this.color = color;
    // NOTE: Returning this for chaining
    return this;
  }

  save() {
    console.log(this.make, this.model, this.color);
    // NOTE: Returning this for chaining
    return this;
  }
}

const car = new Car("Ford", "F-150", "red").setColor("pink").save();
```