A Gentle Introduction to Tabris.js

Rob McCormack P.Eng.

Table of Contents

Tabris.js: A Gentle Introduction

Introduction

> Tabris.js is a mobile framework that lets you develop apps for iOS, Android and Windows from a single code base written entirely in JavaScript or TypeScript and JSX.

A Gentle Introduction to Tabris.js will let you explore and learn Tabris.js

First Steps

You can try out Tabris.js without installing anything on your computer.

- Install the Tabris.js Developer App on your device and browse through the included examples.
- Play with the JavaScript code of a simple Tabris.js app online in the Tabris.js Playground .
- Load your edited version in the Developer App by scanning the bar code on the playground page.

To start developing a real Tabris.js app, follow theQuick Start Guide. We also have an excellentebookthat explains how to create, deploy and test your first Tabris.js app.

A Gentle Introduction to Tabris.js

Rob McCormack P.Eng.

#Your First Tabris JS App

and your second

Test Link: https://github.com/mrmccormack/tabris-gitbook/blob/master/testfromgithub.js

```
const {ui, ImageView, AlertDialog, Button} = require('tabris');
// example of images side by side - Matt.

const IMAGE_PATH = 'https://mrmccormack.github.io/imd-learning-t
abris/images/';
const DICE_OFFSET = 30;

// long pressing will enable/disable cheatMode (doubles to roll
all the time)
let cheatMode = false;

let diceImage1 = new ImageView({
  centerY:0,
  centerX: -DICE_OFFSET,
  image: IMAGE_PATH + '6.png'
}).appendTo(ui.contentView);

let diceImage2 = new ImageView({
  centerY:0,
  centerX: DICE_OFFSET,
  image: IMAGE_PATH + '1.png'
}).appendTo(ui.contentView);



// event outsite create new
diceImage1.on('tap', () => {
console.log ('diceImage1');

if (cheatMode) {
```

```
var rand = 1 + Math.floor(Math.random() * 6);
diceImage1.image = IMAGE_PATH + rand + '.png';
diceImage2.image = IMAGE_PATH + rand + '.png';
} else {
var rand = 1 + Math.floor(Math.random() * 6);
diceImage1.image = IMAGE_PATH + rand + '.png';
var rand = 1 + Math.floor(Math.random() * 6);
diceImage2.image = IMAGE_PATH + rand + '.png';
}

})

diceImage2.on('tap', () => {
console.log ('diceImage2');
if (cheatMode) {

var rand = 1 + Math.floor(Math.random() * 6);
diceImage1.image = IMAGE_PATH + rand + '.png';
diceImage2.image = IMAGE_PATH + rand + '.png';
} else {
var rand = 1 + Math.floor(Math.random() * 6);
diceImage1.image = IMAGE_PATH + rand + '.png';
var rand = 1 + Math.floor(Math.random() * 6);
diceImage2.image = IMAGE_PATH + rand + '.png';
}
  })

diceImage1.on('longpress', () => {
console.log ('Entering Cheat Mode - Good luck');
cheatMode = true;  // toggle ???
  })

diceImage2.on('longpress', () => {
console.log ('Leaving Cheat Mode - Good luck');
cheatMode = false;  // toggle ???
  })
```

```
let btnShowDice = new Button({
  centerX: 0,
  top: 'prev() 10',
  text: 'Show / Hide  dice'
})
.on('select', () => {
diceImage1.visible = !diceImage1.visible;
diceImage2.visible = !diceImage2.visible;

}).appendTo(ui.contentView);


let btnOpacityDice = new Button({
  centerX: 0,
  top: 'prev() 10',
  text: 'Change Opacity'
})
.on('select', () => {
diceImage1.opacity = 0.5;

}).appendTo(ui.contentView);
```

This is a test asdf

Now is the time

Naming Conventions

Widgets

Generally, if there is on only one Widget in an app, just use the full name

`txtMarkDown`

Naming conventions for variables, constants, functions and classes

TL;DR: Use ***lowerCamelCase*** when naming constants, variables and functions and ***UpperCamelCase*** (capital first letter as well) when naming classes. This will help you to easily distinguish between plain variables / functions, and classes that require instantiation. Use descriptive names, but try to keep them short.

Otherwise: Javascript is the only language in the world which allows to invoke a constructor ("Class") directly without instantiating it first. Consequently, Classes and function-constructors are differentiated by starting with UpperCamelCase.

Code Example

```
// for class name we use UpperCamelCase
class SomeClassExample {}

// for const names we use the const keyword and lowerCamelCase
const config = {
  key: 'value'
};

// for variables and functions names we use lowerCamelCase
let someVariableExample = 'value';
function doSomething() {}
```

variables, constants etc.

- This is nice (string literals or integer literals):

```
const PI = 3.14;
const ADDRESS = '10.0.0.1';
```

> but...

```
const myObject = {'key': 'value'};
const userSuppliedNumber = getInputNumber()
```

NOTE:

- Google JavaScript Style Guide says:
- Declare all local variables with either const or let. Use const by default, unless a variable needs to be reassigned. The var keyword must not be used.

```
// Create the activity indicator centered in the page
let activityIndicator = new ActivityIndicator({
  centerX: 0,
  centerY: 0
}).appendTo(ui.contentView);
```

```
let txiMarkDown = new TextInput({
  left: 8, right: 8, top: 'prev() 10',
  height: 100,
  message: MESSAGE,
  type: 'multiline',
  text: INITIAL_TEXT
}).appendTo(ui.contentView);
```

Widget	Prefix	Example
TextInput	**txi**	**txiMarkDown**
TextView	**txv**	**txvCountry**
Button	**btn**	**btnReadFile**

`txtMarkDown`

```
// Create the activity indicator centered in the page
let activityIndicator = new ActivityIndicator({
  centerX: 0,
  centerY: 0
}).appendTo(ui.contentView);
```

```
let txiMarkDown = new TextInput({
  left: 8, right: 8, top: 'prev() 10',
  height: 100,
  message: MESSAGE,
  type: 'multiline',
  text: INITIAL_TEXT
}).appendTo(ui.contentView);
```

`Widget`	`Prefix`	`Example`
`TextInput`	`txi`	`txiMarkDown`
`TextView`	`txv`	`txvCountry`
`Button`	`btn`	`btnReadFile`
asdf	asdf	asdf

Ordering of Widget properties

Use of layoutData

What is the advantage?

1. it validates against Javascript Standard on ONE line.
2. what order, width height always last.

```
layoutData: {left: offset, top: offset, width: 100, height: 100}
,
```

Widget properties take the form

```
property: parameter,
```

Note commas separate properties as shown:

```
left: 8,
```

Properties may be listed on multiple lines or on one line:

```
let txiMarkDown = new TextInput({
  left: 8, right: 8, top: 'prev() 10',
  height: 100,
  message: 'Enter URL here...',
  type: 'multiline',
  text: INITIAL_TEXT
}).appendTo(ui.contentView);
```

For readability, this book sets all positioning properties on one line, immediately afgter the `let` statement

```
left: 8, right: 8, top: 'prev() 10'
```

Dimensions are shown next

```
height: 100,
```

You might list the rest of properties in order of importance, although that might be difficult to decide upon.

Coding Style

- We will follow the *JavaScript Standard Style* for all Tabris.js code.

This is a TL;DR of the standard JavaScript rules.

StandardJS — The Rules in Brief

- **2 spaces** – for indentation
- **Single quotes for strings** – except to avoid escaping
- **No unused variables** – this one catches *tons* of bugs!
- **No semicolons** – It's fine. Really!
- **Never start a line with `(`, `[`, or `` ` ``**
    - This is the **only** gotcha with omitting semicolons – *automatically checked for you!*
    - More details
- **Space after keywords** `if (condition) { ... }`
- **Space after function name** `function name (arg) { ... }`
- Always use `===` instead of `==` – but `obj == null` is allowed to check `null || undefined`.
- Always handle the node.js `err` function parameter
- Always prefix browser globals with `window` – except `document` and `navigator` are okay
    - Prevents accidental use of poorly-named browser globals like `open`, `length`, `event`, and `name`.
- **And more goodness** – *give* `standard` *a try today!*

Open Source Supporters

Rules in Detail

- **Use 2 spaces** for indentation.

  eslint: `indent`

  ```
  function hello (name) {
    console.log('hi', name)
  }
  ```

- **Use single quotes for strings** except to avoid escaping.

  eslint: `quotes`

  ```
  console.log('hello there')
  $("<div class='box'>")
  ```

- **No unused variables.**

  eslint: `no-unused-vars`

  ```
  function myFunction () {
    var result = something()   // ✗ avoid
  }
  ```

- **Add a space after keywords.**

  eslint: `keyword-spacing`

  ```
  if (condition) { ... }   // ✓ ok
  if(condition) { ... }    // ✗ avoid
  ```

- **Add a space before a function declaration's parentheses.**

  eslint: `space-before-function-paren`

  ```
  function name (arg) { ... }    // ✓ ok
  function name(arg) { ... }     // ✗ avoid

  run(function () { ... })       // ✓ ok
  run(function() { ... })        // ✗ avoid
  ```

- **Always use** `===` instead of `==`.
  Exception: `obj == null` is allowed to check for `null || undefined`.

  eslint: `eqeqeq`

  ```
  if (name === 'John')   // ✓ ok
  if (name == 'John')    // ✗ avoid
  ```

  ```
  if (name !== 'John')   // ✓ ok
  if (name != 'John')    // ✗ avoid
  ```

- **Infix operators** must be spaced.

  eslint: `space-infix-ops`

  ```
  // ✓ ok
  var x = 2
  var message = 'hello, ' + name + '!'
  ```

  ```
  // ✗ avoid
  var x=2
  var message = 'hello, '+name+'!'
  ```

- **Commas should have a space** after them.

  eslint: `comma-spacing`

```
// ✓ ok
var list = [1, 2, 3, 4]
function greet (name, options) { ... }
```

```
// ✗ avoid
var list = [1,2,3,4]
function greet (name,options) { ... }
```

- **Keep else statements** on the same line as their curly braces.

  eslint: `brace-style`

```
// ✓ ok
if (condition) {
  // ...
} else {
  // ...
}
```

```
// ✗ avoid
if (condition) {
  // ...
}
else {
  // ...
}
```

- **For multi-line if statements,** use curly braces.

  eslint: `curly`

```
// ✓ ok
if (options.quiet !== true) console.log('done')
```

```js
// ✓ ok
if (options.quiet !== true) {
  console.log('done')
}
```

```js
// ✗ avoid
if (options.quiet !== true)
  console.log('done')
```

- **Always handle the `err` function parameter.**

  eslint: `handle-callback-err`

  ```js
  // ✓ ok
  run(function (err) {
    if (err) throw err
    window.alert('done')
  })
  ```

  ```js
  // ✗ avoid
  run(function (err) {
    window.alert('done')
  })
  ```

- **Always prefix browser globals** with `window.` .
  Exceptions are: `document` , `console` and `navigator` .

  eslint: `no-undef`

  ```js
  window.alert('hi')   // ✓ ok
  ```

- **Multiple blank lines not allowed.**

  eslint: `no-multiple-empty-lines`

```
// ✓ ok
var value = 'hello world'
console.log(value)
```

```js // ✗ avoid var value = 'hello world'

console.log(value)

```
* **For the ternary operator** in a multi-line setting, place `?
` and `:` on their own lines.

  eslint: [`operator-linebreak`](http://eslint.org/docs/rules/op
erator-linebreak)

  ```js
  // ✓ ok
  var location = env.development ? 'localhost' : 'www.api.com'

  // ✓ ok
  var location = env.development
    ? 'localhost'
    : 'www.api.com'

  // ✗ avoid
  var location = env.development ?
    'localhost' :
    'www.api.com'
```

- **For var declarations,** write each declaration in its own statement.

  eslint: `one-var`

```
// ✓ ok
var silent = true
var verbose = true

// ✗ avoid
var silent = true, verbose = true

// ✗ avoid
var silent = true,
    verbose = true
```

- **Wrap conditional assignments** with additional parentheses. This makes it clear that the expression is intentionally an assignment (`=`) rather than a typo for equality (`===`).

  eslint: `no-cond-assign`

```
// ✓ ok
while ((m = text.match(expr))) {
  // ...
}

// ✗ avoid
while (m = text.match(expr)) {
  // ...
}
```

- **Add spaces inside single line blocks.**

  eslint: `block-spacing`

```
  function foo () {return true}    // ✗ avoid
  function foo () { return true }  // ✓ ok
```

- **Use camelcase when naming variables and functions.**

  eslint: `camelcase`

```js
function my_function () { }    // ✗ avoid
function myFunction () { }     // ✓ ok

var my_var = 'hello'           // ✗ avoid
var myVar = 'hello'            // ✓ ok
```

- **Trailing commas not allowed.**

  eslint: `comma-dangle`

```js
var obj = {
  message: 'hello',   // ✗ avoid
}
```

- **Commas must be placed at the end of the current line.**

  eslint: `comma-style`

```js
var obj = {
  foo: 'foo'
  ,bar: 'bar'   // ✗ avoid
}

var obj = {
  foo: 'foo',
  bar: 'bar'   // ✓ ok
}
```

- **Dot should be on the same line as property.**

  eslint: `dot-location`

```js
console.
  log('hello')  // ✗ avoid

console
  .log('hello') // ✓ ok
```

- **Files must end with a newline.**

  eslint: `eol-last`

- **No space between function identifiers and their invocations.**

  eslint: `func-call-spacing`

  ```
  console.log ('hello') // ✗ avoid
  console.log('hello')  // ✓ ok
  ```

- **Add space between colon and value in key value pairs.**

  eslint: `key-spacing`

  ```
  var obj = { 'key' : 'value' }    // ✗ avoid
  var obj = { 'key' :'value' }     // ✗ avoid
  var obj = { 'key':'value' }      // ✗ avoid
  var obj = { 'key': 'value' }     // ✓ ok
  ```

- **Constructor names must begin with a capital letter.**

  eslint: `new-cap`

  ```
  function animal () {}
  var dog = new animal()    // ✗ avoid

  function Animal () {}
  var dog = new Animal()    // ✓ ok
  ```

- **Constructor with no arguments must be invoked with parentheses.**

  eslint: `new-parens`

  ```
  function Animal () {}
  var dog = new Animal    // ✗ avoid
  var dog = new Animal()  // ✓ ok
  ```

- **Objects must contain a getter when a setter is defined.**

eslint: `accessor-pairs`

```
var person = {
  set name (value) {    // ✗ avoid
    this.name = value
  }
}

var person = {
  set name (value) {
    this.name = value
  },
  get name () {         // ✓ ok
    return this.name
  }
}
```

- **Constructors of derived classes must call `super`.**

  eslint: `constructor-super`

```
class Dog {
  constructor () {
    super()   // ✗ avoid
  }
}

class Dog extends Mammal {
  constructor () {
    super()   // ✓ ok
  }
}
```

- **Use array literals instead of array constructors.**

  eslint: `no-array-constructor`

```
var nums = new Array(1, 2, 3)   // ✗ avoid
var nums = [1, 2, 3]            // ✓ ok
```

- **Avoid using `arguments.callee` and `arguments.caller` .**

  eslint: `no-caller`

```
function foo (n) {
  if (n <= 0) return

  arguments.callee(n - 1)   // ✗ avoid
}

function foo (n) {
  if (n <= 0) return

  foo(n - 1)
}
```

- **Avoid modifying variables of class declarations.**

  eslint: `no-class-assign`

```
class Dog {}
Dog = 'Fido'    // ✗ avoid
```

- **Avoid modifying variables declared using `const` .**

  eslint: `no-const-assign`

```
const score = 100
score = 125       // ✗ avoid
```

- **Avoid using constant expressions in conditions (except loops).**

  eslint: `no-constant-condition`

```
if (false) {    // ✗ avoid
  // ...
}

if (x === 0) {  // ✓ ok
  // ...
}

while (true) {  // ✓ ok
  // ...
}
```

- **No control characters in regular expressions.**

  eslint: `no-control-regex`

  ```
  var pattern = /\x1f/    // ✗ avoid
  var pattern = /\x20/    // ✓ ok
  ```

- **No `debugger` statements.**

  eslint: `no-debugger`

  ```
  function sum (a, b) {
    debugger      // ✗ avoid
    return a + b
  }
  ```

- **No `delete` operator on variables.**

  eslint: `no-delete-var`

  ```
  var name
  delete name     // ✗ avoid
  ```

- **No duplicate arguments in function definitions.**

  eslint: `no-dupe-args`

```
function sum (a, b, a) {  // ✗ avoid
  // ...
}

function sum (a, b, c) {  // ✓ ok
  // ...
}
```

- **No duplicate name in class members.**

  eslint: `no-dupe-class-members`

  ```
  class Dog {
    bark () {}
    bark () {}    // ✗ avoid
  }
  ```

- **No duplicate keys in object literals.**

  eslint: `no-dupe-keys`

  ```
  var user = {
    name: 'Jane Doe',
    name: 'John Doe'     // ✗ avoid
  }
  ```

- **No duplicate `case` labels in `switch` statements.**

  eslint: `no-duplicate-case`

  ```
  switch (id) {
    case 1:
      // ...
    case 1:     // ✗ avoid
  }
  ```

- **Use a single import statement per module.**

eslint: `no-duplicate-imports`

```js
import { myFunc1 } from 'module'
import { myFunc2 } from 'module'          // ✗ avoid

import { myFunc1, myFunc2 } from 'module' // ✓ ok
```

- **No empty character classes in regular expressions.**

  eslint: `no-empty-character-class`

  ```js
  const myRegex = /^abc[]/      // ✗ avoid
  const myRegex = /^abc[a-z]/   // ✓ ok
  ```

- **No empty destructuring patterns.**

  eslint: `no-empty-pattern`

  ```js
  const { a: {} } = foo         // ✗ avoid
  const { a: { b } } = foo      // ✓ ok
  ```

- **No using `eval()`.**

  eslint: `no-eval`

  ```js
  eval( "var result = user." + propName ) // ✗ avoid
  var result = user[propName]             // ✓ ok
  ```

- **No reassigning exceptions in `catch` clauses.**

  eslint: `no-ex-assign`

```
try {
  // ...
} catch (e) {
  e = 'new value'             // ✗ avoid
}

try {
  // ...
} catch (e) {
  const newVal = 'new value'  // ✓ ok
}
```

- **No extending native objects.**

  eslint: `no-extend-native`

```
Object.prototype.age = 21     // ✗ avoid
```

- **Avoid unnecessary function binding.**

  eslint: `no-extra-bind`

```
const name = function () {
  getName()
}.bind(user)    // ✗ avoid

const name = function () {
  this.getName()
}.bind(user)    // ✓ ok
```

- **Avoid unnecessary boolean casts.**

  eslint: `no-extra-boolean-cast`

```
const result = true
if (!!result) {   // ✗ avoid
  // ...
}

const result = true
if (result) {     // ✓ ok
  // ...
}
```

- **No unnecessary parentheses around function expressions.**

  eslint: `no-extra-parens`

  ```
  const myFunc = (function () { })   // ✗ avoid
  const myFunc = function () { }     // ✓ ok
  ```

- **Use `break` to prevent fallthrough in `switch` cases.**

  eslint: `no-fallthrough`

```
switch (filter) {
  case 1:
    doSomething()    // ✗ avoid
  case 2:
    doSomethingElse()
}

switch (filter) {
  case 1:
    doSomething()
    break           // ✓ ok
  case 2:
    doSomethingElse()
}

switch (filter) {
  case 1:
    doSomething()
    // fallthrough  // ✓ ok
  case 2:
    doSomethingElse()
}
```

- **No floating decimals.**

  eslint: `no-floating-decimal`

  ```
  const discount = .5      // ✗ avoid
  const discount = 0.5     // ✓ ok
  ```

- **Avoid reassigning function declarations.**

  eslint: `no-func-assign`

  ```
  function myFunc () { }
  myFunc = myOtherFunc    // ✗ avoid
  ```

- **No reassigning read-only global variables.**

eslint: `no-global-assign`

```
window = {}      // ✗ avoid
```

- **No implied `eval()` .**

  eslint: `no-implied-eval`

```
setTimeout("alert('Hello world')")                   // ✗ av
oid
setTimeout(function () { alert('Hello world') })     // ✓ ok
```

- **No function declarations in nested blocks.**

  eslint: `no-inner-declarations`

```
if (authenticated) {
  function setAuthUser () {}    // ✗ avoid
}
```

- **No invalid regular expression strings in `RegExp` constructors.**

  eslint: `no-invalid-regexp`

```
RegExp('[a-z')    // ✗ avoid
RegExp('[a-z]')   // ✓ ok
```

- **No irregular whitespace.**

  eslint: `no-irregular-whitespace`

```
function myFunc () /*<NBSP>*/{}   // ✗ avoid
```

- **No using `__iterator__` .**

  eslint: `no-iterator`

```
Foo.prototype.__iterator__ = function () {}   // ✗ avoid
```

- **No labels that share a name with an in scope variable.**

  eslint: `no-label-var`

  ```
  var score = 100
  function game () {
    score: 50         // ✗ avoid
  }
  ```

- **No label statements.**

  eslint: `no-labels`

  ```
  label:
    while (true) {
      break label     // ✗ avoid
    }
  ```

- **No unnecessary nested blocks.**

  eslint: `no-lone-blocks`

  ```
  function myFunc () {
    {                   // ✗ avoid
      myOtherFunc()
    }
  }

  function myFunc () {
    myOtherFunc()       // ✓ ok
  }
  ```

- **Avoid mixing spaces and tabs for indentation.**

  eslint: `no-mixed-spaces-and-tabs`

- **Do not use multiple spaces except for indentation.**

  eslint: `no-multi-spaces`

  ```
  const id =    1234    // ✗ avoid
  const id = 1234       // ✓ ok
  ```

- **No multiline strings.**

  eslint: `no-multi-str`

  ```
  const message = 'Hello \
                   world'     // ✗ avoid
  ```

- **No `new` without assigning object to a variable.**

  eslint: `no-new`

  ```
  new Character()                     // ✗ avoid
  const character = new Character()   // ✓ ok
  ```

- **No using the `Function` constructor.**

  eslint: `no-new-func`

  ```
  var sum = new Function('a', 'b', 'return a + b')    // ✗ avo
  id
  ```

- **No using the `Object` constructor.**

  eslint: `no-new-object`

  ```
  let config = new Object()   // ✗ avoid
  ```

- **No using `new require`.**

  eslint: `no-new-require`

```
const myModule = new require('my-module')    // ✗ avoid
```

- **No using the `Symbol` constructor.**

  eslint: `no-new-symbol`

  ```
  const foo = new Symbol('foo')   // ✗ avoid
  ```

- **No using primitive wrapper instances.**

  eslint: `no-new-wrappers`

  ```
  const message = new String('hello')   // ✗ avoid
  ```

- **No calling global object properties as functions.**

  eslint: `no-obj-calls`

  ```
  const math = Math()   // ✗ avoid
  ```

- **No octal literals.**

  eslint: `no-octal`

  ```
  const num = 042     // ✗ avoid
  const num = '042'   // ✓ ok
  ```

- **No octal escape sequences in string literals.**

  eslint: `no-octal-escape`

  ```
  const copyright = 'Copyright \251'  // ✗ avoid
  ```

- **Avoid string concatenation when using `__dirname` and `__filename`.**

  eslint: `no-path-concat`

```
const pathToFile = __dirname + '/app.js'            // ✗ avo
id
const pathToFile = path.join(__dirname, 'app.js')   // ✓ ok
```

- **Avoid using `__proto__`. Use** `getPrototypeOf` instead.

  eslint: `no-proto`

  ```
  const foo = obj.__proto__               // ✗ avoid
  const foo = Object.getPrototypeOf(obj)  // ✓ ok
  ```

- **No redeclaring variables.**

  eslint: `no-redeclare`

  ```
  let name = 'John'
  let name = 'Jane'     // ✗ avoid

  let name = 'John'
  name = 'Jane'         // ✓ ok
  ```

- **Avoid multiple spaces in regular expression literals.**

  eslint: `no-regex-spaces`

  ```
  const regexp = /test   value/   // ✗ avoid

  const regexp = /test {3}value/  // ✓ ok
  const regexp = /test value/     // ✓ ok
  ```

- **Assignments in return statements must be surrounded by parentheses.**

  eslint: `no-return-assign`

```
function sum (a, b) {
  return result = a + b     // ✗ avoid
}

function sum (a, b) {
  return (result = a + b)   // ✓ ok
}
```

- **Avoid assigning a variable to itself**

  eslint: `no-self-assign`

  ```
  name = name   // ✗ avoid
  ```

- **Avoid comparing a variable to itself.**

  eslint: `no-self-compare`

  ```
  if (score === score) {}   // ✗ avoid
  ```

- **Avoid using the comma operator.**

  eslint: `no-sequences`

  ```
  if (doSomething(), !!test) {}   // ✗ avoid
  ```

- **Restricted names should not be shadowed.**

  eslint: `no-shadow-restricted-names`

  ```
  let undefined = 'value'     // ✗ avoid
  ```

- **Sparse arrays are not allowed.**

  eslint: `no-sparse-arrays`

```
let fruits = ['apple',, 'orange']       // ✗ avoid
```

- **Tabs should not be used**

  eslint: `no-tabs`

- **Regular strings must not contain template literal placeholders.**

  eslint: `no-template-curly-in-string`

```
const message = 'Hello ${name}'   // ✗ avoid
const message = `Hello ${name}`   // ✓ ok
```

- **`super()` must be called before using `this`.**

  eslint: `no-this-before-super`

```
class Dog extends Animal {
  constructor () {
    this.legs = 4     // ✗ avoid
    super()
  }
}
```

- **Only `throw` an `Error` object.**

  eslint: `no-throw-literal`

```
throw 'error'                // ✗ avoid
throw new Error('error')     // ✓ ok
```

- **Whitespace not allowed at end of line.**

  eslint: `no-trailing-spaces`

- **Initializing to `undefined` is not allowed.**

  eslint: `no-undef-init`

```js
let name = undefined    // ✗ avoid

let name
name = 'value'          // ✓ ok
```

- **No unmodified conditions of loops.**

  eslint: `no-unmodified-loop-condition`

```js
for (let i = 0; i < items.length; j++) {...}    // ✗ avoid
for (let i = 0; i < items.length; i++) {...}    // ✓ ok
```

- **No ternary operators when simpler alternatives exist.**

  eslint: `no-unneeded-ternary`

```js
let score = val ? val : 0     // ✗ avoid
let score = val || 0          // ✓ ok
```

- **No unreachable code after `return`, `throw`, `continue`, and `break` statements.**

  eslint: `no-unreachable`

```js
function doSomething () {
  return true
  console.log('never called')     // ✗ avoid
}
```

- **No flow control statements in `finally` blocks.**

  eslint: `no-unsafe-finally`

```
try {
  // ...
} catch (e) {
  // ...
} finally {
  return 42     // ✗ avoid
}
```

- **The left operand of relational operators must not be negated.**

  eslint: `no-unsafe-negation`

  ```
  if (!key in obj) {}       // ✗ avoid
  if (!(key in obj)) {}     // ✓ ok
  ```

- **Avoid unnecessary use of `.call()` and `.apply()`.**

  eslint: `no-useless-call`

  ```
  sum.call(null, 1, 2, 3)   // ✗ avoid
  ```

- **Avoid using unnecessary computed property keys on objects.**

  eslint: `no-useless-computed-key`

  ```
  const user = { ['name']: 'John Doe' }   // ✗ avoid
  const user = { name: 'John Doe' }       // ✓ ok
  ```

- **No unnecessary constructor.**

  eslint: `no-useless-constructor`

  ```
  class Car {
    constructor () {      // ✗ avoid
    }
  }
  ```

- **No unnecessary use of escape.**

  eslint: `no-useless-escape`

  ```
  let message = 'Hell\o'  // ✗ avoid
  ```

- **Renaming import, export, and destructured assignments to the same name is not allowed.**

  eslint: `no-useless-rename`

  ```
  import { config as config } from './config'     // ✗ avoid
  import { config } from './config'               // ✓ ok
  ```

- **No whitespace before properties.**

  eslint: `no-whitespace-before-property`

  ```
  user .name      // ✗ avoid
  user.name       // ✓ ok
  ```

- **No using `with` statements.**

  eslint: `no-with`

  ```
  with (val) {...}    // ✗ avoid
  ```

- **Maintain consistency of newlines between object properties.**

  eslint: `object-property-newline`

```
const user = {
  name: 'Jane Doe', age: 30,
  username: 'jdoe86'            // ✗ avoid
}

const user = { name: 'Jane Doe', age: 30, username: 'jdoe86'
 }     // ✓ ok

const user = {
  name: 'Jane Doe',
  age: 30,
  username: 'jdoe86'
}
      // ✓ ok
```

- **No padding within blocks.**

  eslint: `padded-blocks`

```
if (user) {
                            // ✗ avoid
  const name = getName()

}

if (user) {
  const name = getName()    // ✓ ok
}
```

- **No whitespace between spread operators and their expressions.**

  eslint: `rest-spread-spacing`

```
fn(... args)    // ✗ avoid
fn(...args)     // ✓ ok
```

- **Semicolons must have a space after and no space before.**

eslint: `semi-spacing`

```
for (let i = 0 ;i < items.length ;i++) {...}    // ✗ avoid
for (let i = 0; i < items.length; i++) {...}    // ✓ ok
```

- **Must have a space before blocks.**

  eslint: `space-before-blocks`

  ```
  if (admin){...}     // ✗ avoid
  if (admin) {...}    // ✓ ok
  ```

- **No spaces inside parentheses.**

  eslint: `space-in-parens`

  ```
  getName( name )     // ✗ avoid
  getName(name)       // ✓ ok
  ```

- **Unary operators must have a space after.**

  eslint: `space-unary-ops`

  ```
  typeof!admin        // ✗ avoid
  typeof !admin        // ✓ ok
  ```

- **Use spaces inside comments.**

  eslint: `spaced-comment`

  ```
  //comment           // ✗ avoid
  // comment          // ✓ ok

  /*comment*/         // ✗ avoid
  /* comment */       // ✓ ok
  ```

- **No spacing in template strings.**

eslint: `template-curly-spacing`

```
const message = `Hello, ${ name }`     // ✗ avoid
const message = `Hello, ${name}`       // ✓ ok
```

- **Use `isNaN()` when checking for `NaN`.**

  eslint: `use-isnan`

  ```
  if (price === NaN) { }       // ✗ avoid
  if (isNaN(price)) { }        // ✓ ok
  ```

- **`typeof` must be compared to a valid string.**

  eslint: `valid-typeof`

  ```
  typeof name === 'undefimed'     // ✗ avoid
  typeof name === 'undefined'     // ✓ ok
  ```

- **Immediately Invoked Function Expressions (IIFEs) must be wrapped.**

  eslint: `wrap-iife`

  ```
  const getName = function () { }()     // ✗ avoid

  const getName = (function () { }())   // ✓ ok
  const getName = (function () { })()   // ✓ ok
  ```

- **The `*` in `yield*` expressions must have a space before and after.**

  eslint: `yield-star-spacing`

  ```
  yield* increment()    // ✗ avoid
  yield * increment()   // ✓ ok
  ```

- **Avoid Yoda conditions.**

  eslint: `yoda`

```
if (42 === age) { }    // ✗ avoid
if (age === 42) { }    // ✓ ok
```

Semicolons

- No semicolons. (see: 1, 2, 3)

  eslint: `semi`

  ```
  window.alert('hi')   // ✓ ok
  window.alert('hi');  // ✗ avoid
  ```

- Never start a line with `(`, `[`, or `` ` ``. This is the only gotcha with omitting semicolons, and standard protects you from this potential issue.

  eslint: `no-unexpected-multiline`

  ```
  // ✓ ok
  ;(function () {
    window.alert('ok')
  }())

  // ✗ avoid
  (function () {
    window.alert('ok')
  }())
  ```

  ```
  // ✓ ok
  ;[1, 2, 3].forEach(bar)

  // ✗ avoid
  [1, 2, 3].forEach(bar)
  ```

```
// ✓ ok
;`hello`.indexOf('o')

// ✗ avoid
`hello`.indexOf('o')
```

Note: If you're often writing code like this, you may be trying to be too clever.

Clever short-hands are discouraged, in favor of clear and readable expressions, whenever possible.

Instead of this:

```
;[1, 2, 3].forEach(bar)
```

This is strongly preferred:

```
var nums = [1, 2, 3]
nums.forEach(bar)
```

Helpful reading

- An Open Letter to JavaScript Leaders Regarding Semicolons
- JavaScript Semicolon Insertion – Everything you need to know

And a helpful video:

- Are Semicolons Necessary in JavaScript? - YouTube

All popular code minifiers in use today use AST-based minification, so they can handle semicolon-less JavaScript with no issues (since semicolons are not required in JavaScript).

Excerpt from *"An Open Letter to JavaScript Leaders Regarding Semicolons"*:

> [Relying on automatic semicolon insertion] is quite safe, and perfectly valid JS that every browser understands. Closure compiler, yuicompressor, packer, and jsmin all can properly minify it. There is no performance impact anywhere.
>
> I am sorry that, instead of educating you, the leaders in this language community have given you lies and fear. That was shameful. I recommend learning how statements in JS are actually terminated (and in which cases they are not terminated), so that you can write code that you find beautiful.
>
> In general, `\n` ends a statement unless:
>
> 1. The statement has an unclosed paren, array literal, or object literal or ends in some other way that is not a valid way to end a statement. (For instance, ending with `.` or `,`.)
> 2. The line is `--` or `++` (in which case it will decrement/increment the next token.)
> 3. It is a `for()`, `while()`, `do`, `if()`, or `else`, and there is no `{`
> 4. The next line starts with `[`, `(`, `+`, `*`, `/`, `-`, `,`, `.`, or some other binary operator that can only be found between two tokens in a single expression.
>
> The first is pretty obvious. Even JSLint is ok with `\n` chars in JSON and parenthesized constructs, and with `var` statements that span multiple lines ending in `,`.
>
> The second is super weird. I've never seen a case (outside of these sorts of conversations) where you'd want to do write `i\n++\nj`, but, point of fact, that's parsed as `i; ++j`, not `i++; j`.
>
> The third is well understood, if generally despised. `if (x)\ny()` is equivalent to `if (x) { y() }`. The construct doesn't end until it reaches either a block, or a statement.
>
> `;` is a valid JavaScript statement, so `if(x);` is equivalent to `if(x){}` or, "If x, do nothing." This is more commonly applied to loops where the loop check also is the update function. Unusual, but not unheard of.

> The fourth is generally the fud-inducing “oh noes, you need semicolons!” case. But, as it turns out, it’s quite easy to *prefix* those lines with semicolons if you don’t mean them to be continuations of the previous line. For example, instead of this:
>
> ```
> foo();
> [1,2,3].forEach(bar);
> ```
>
> you could do this:
>
> ```
> foo()
> ;[1,2,3].forEach(bar)
> ```
>
> The advantage is that the prefixes are easier to notice, once you are accustomed to never seeing lines starting with `(` or `[` without semis.

Big Arrow Functions

> It’s Time to Embrace Arrow Functions

- https://medium.com/javascript-scene/familiarity-bias-is-holding-you-back-its-time-to-embrace-arrow-functions-3d37e1a9bb75

Event Handling Style

Events may be shown in two ways.

Events as part of declaration

```
// button with event outside
let resetbutton = new Button({
    centerX: 0,
    top: 'prev() 10',
    text: '  Reset'

  })
.appendTo(ui.contentView);

// event outsite create new
resetbutton.on('select', () => {
console.log ('you pressed reset');
console.log (urlInput.text);
  let HTML_TEMPLATE =  '<!DOCTYPE html>\
<html>\
<title>Hello Strapdown</title>\
<xmp theme="united" style="display:none;">\
' + urlInput.text + '</xmp>\
<script src="http://strapdownjs.com/v/0.2/strapdown.js"></script
>\
</html>'

 webView.html = HTML_TEMPLATE;
  })
```

Events outside of declaration

This book always separates the event from the declaration for readability.

Promises Explained

- explain them, in Tabris.js

Sample of lost of .js code

```
const {Button, TextView, TextInput, WebView, ui} = require('tabr
is');

const INITIAL_TEXT = '![](https://cdn2.iconfinder.com/data/icons
/nodejs-1/64/nodejs-64.png)\n# Heading \n- one \n- two \n---\n \
Instant and elegant Markdown documents\n```js\n \
alert("Hello")\; \n```' ;

const MESSAGE = 'Enter Markdown...';
const TITLE = 'Strapdown.js';

new TextView({
  left: 8, right: 8, top: 16,
  text: 'Enter some Markdown:',
}).appendTo(ui.contentView);

let txiMarkDown = new TextInput({
  left: 8, right: 8, top: 'prev() 10',
  height: 100,
  message: MESSAGE,
  type: 'multiline',
  text: INITIAL_TEXT
}).appendTo(ui.contentView);

let btnRenderMarkDown = new Button({
    centerX: 0, top: 'prev() 10',
    text: 'Render Markdown'
  })
.appendTo(ui.contentView);

btnRenderMarkDown.on('select', () => {
  renderMarkdown();
})

// This will render each time text is changed
```

```
// txiMarkDown.on('textChanged', () => {
//   renderMarkdown();
// })

let webView = new WebView({
  left: 0, top: 'prev() 8', right: 0, bottom: 0
}).appendTo(ui.contentView);

function renderMarkdown() {
  let HTML_TEMPLATE =  '<!DOCTYPE html>\
  <html>\
  <title>' + TITLE + '</title>\
  <xmp theme="Cerulean" style="display:none;">\
  ' + txiMarkDown.text + '</xmp>\
  <script src="http://strapdownjs.com/v/0.2/strapdown.js"></scri
pt>\
  </html>';
  webView.html = HTML_TEMPLATE;
}

// render when loaded
renderMarkdown();
```

Reference for using GitBook

- you can edit raw Markdown, by :
- On the riht bottom corner you can see a question mark inside there is a menu. click the "edit markdows" entry.

Plugins

> It seems that plugins only work locally... that would explain the trouble.
>
> - can you embed a gist or a File from repository - possible, but Rob hasn't got it working yet
> - Don't use Plugins, could not get the embed code (use just a link to updated version) and Dark Theme didn't work either.

How to do a cover.

- not sure of this one, you can include cover.jpg
- Ref:
