Rithm Code Requirements: JavaScript

> IMPORTANT **We expect you to meet these requirements!**
>
> Good code style is an important thing for professional developers, and most companies will have strict coding guidelines.
>
> Read this carefully and keep a copy nearby. We expect your code here to follow these guidelines. We know that its a lot to get used to, but take your time and review these guidelines periodically. Ask us if you have questions.

JavaScript Files

1. **Semicolons** go at the end of lines:

   ```
   let x = 42;
   ```

   but not at end of code blocks:

   ```
   if (x === 42) {
     console.log("ok");
   }  // <- no semicolon here

   function double(x) {
     return x * 2;
   }  // <- no semicolon here
   ```

2. **Keep line lengths to 80-90, max.** Turn off any feature in your editor that "soft-wraps" lines for you; you want to control where line breaks happen so you can use them meaningfully to lead others to understand your code.

   You can add the following *"editor.rulers": [80,90],* to your *settings.json* in VSCode to provide vertical rulers at 80 and 120 characters.

3. **JavaScript file names** should contain no spaces or punctuation except for periods: `generateIds.js`, `generateIds.test.js`.

4. **Strict mode:** `"use strict";` should be the first line of **every** JS file.

5. **Format document:** Use VSCode "Format Document" frequently, and especially before seeking a code review or submitting work: «⌥⇧F».

Documentation and Comments

Writing good documentation for code is **required for all code here.**

1. **Every function and class should get a "docstring"**: this should explain what the function does, what its parameters are, and explain clearly what is returned. This should use `/** ... */` comments and should appear *before* the class or function:

```
/** Return given number `n`, doubled. */

function double(n) {
  return n * 2;
}


/** Calculate area of right triangle:
    -a, b: lengths of opposite and adjacent sides.

    Returns area or undefined for invalid input.
 */

function getArea(a, b) {
  // ... snip ...
}
```

In cases where the returned value is complex or nested, an *example* of it is typically clearer than trying to describe it:

```
/** Searches for given `movieNames` from `movieDataFile` file.
 *
 * Returns [ {title, description, rating}, ... ].
 */

function getMovies(movieNames, movieDataFile) {
  // ... snip ...
}
```

2. **Docstrings are different from normal comments.** Docstrings should not explain how a function works on the inside. If the code inside a function requires comments to explain to the reader how it works, this should use `//` comments, and should be inside the function, above the code it explicates:

```
/** Draw line from x to y, in current color. */

function drawLine(x, y) {
  // Check if user has a high-density display so we can adjust the color
  // if needed.
  // ... snip ...
}
```

3. If there is **commented-out code**, this should always be explained:

```
function getMoviePrice(movieId) {
  // previous formula before we got rid of coupon codes:
  //   let discount = getCouponCode(movieId);
  //   ... snip ...
}
```

4. **Comment markers:** Use *TODO* and *FIXME* markers to indicate incomplete work, or work that has bugs or should be refactored:

```
// TODO: allow other colors than black

// FIXME: doesn't work if number is negative
```

Variables

1. Variables should be in **"camel case"** (*userAge*, not *user_age*).
2. **Using let and const:** Use *let* to define variables if they will be redefined or mutated; if not, use *const*.

3. **Do not use** *var*.
4. Variable names should reflect the purpose/meaning a variable rather than its type. For example, a list of ages should called *ages*, not *array* or *ageArr*.
5. **Variables local to a function** can be short if their purpose is obvious:

```
let age = calcAge(birthDate);
```

6. **Global variables should always have very clear names**, since they can be used anywhere in a program:

```
let usersCurrentAge = calcAge(birthDate);
```

   Generally, avoid the use of global variables (except global constants).

7. **Global constants** are variables that are in the global scope and always the same should have *ALL_CAP_NAMES*:

```
// is always the same and in the global scope
const MAX_SCORE = 1000;

const randomNumber = Math.random();  // not always the same, so normal name
const formAge = document.querySelector("#age-form-age").value;  // same

function myFunction() {
  // not in global scope
  const operatingSystem = ["Windows", "Linux", "MacOS"];
}
```

   These should be written as *randomNumber* and *formAge*.

8. **Don't "shadow variables"** (it's confusing):

```
let userNames = ["elie", "jane"];

function getUsers() {
  // this will be a *different* variable and block getting to the global
  // `userNames` in this function. Don't do this!
  let userNames = ["bob", "joel"];
}
```

9. It's often a good idea to **include units** in variable names:

```
let delay = 1000;  // WRONG
let delayMsec = 1000;  // GOOD

let beachTemp = 90;  // WRONG
let beachTempF = 90;  // GOOD
```

10. **Use conventional names** if appropriate:

```
// `i` is very conventional for the index in an array being looped over
for (let i = 0; i < userNames.length; i += 1) {
  // ... snip ...
}

// but if there's a clearer name, given what you're doing, use that ---
// here, `x` and `y` are clearer as the coordinates of a grid:
let ticTacToeBoard = [ ["X", "X", "O"], ["O", "O", " "], ["O", " ", "X"] ];
for (let y = 0; y < 3; y += 1) {
  for (let x = 0; x < 3; x += 1) {
    // ... snip ...
  }
}
```

1. **Classes** should have name like *ClassName*; normal variable should never start with a capital letter.
2. **jQuery DOM elements** should have names like *$gameBoard*.
3. **Define variables where first needed**, not before:

```
function myFunction(a, b) {
  let found = [];  // WRONG

  // ... lots of code that doesn't use found ...

  for (let x of items) {
    found.push(x);
  }
}
```

Functions

1. **Functions should be small** (typically, less than 20-40 lines) and have a single purpose.
2. **Use blank lines** in a function to show a shift to a different part:

```
function drawLine(x, y, color) {
  const screenX = convertPoint(x);
  const screenY = convertPoint(y);

  changeColor(WHITE);
  clearBox(screenX, screenY);

  changeColor(BLACK);
  plotUsingPoints(screenX, screenY);
}
```

3. **Good function names** are often *verb-noun*:

```
// WRONG: follow the verbNoun convention
function user()

// RIGHT: function name starts with a verb and is followed by a noun
function createUser() { }
```

4. **Separation of concerns**: keep functions about logic in a different place than functions around DOM manipulation or UI (these are often worked on by different teams):

```
/** Get todo from other system; returns text of todo. */

function getTodoById(todoId) {
  // Go to API and find a todo
  // ...
}

/** Display todo text in the list. */

function displayTodo(todo) {
  const todoHTML = document.createElement("li");
  todoHTML.innerText = `${}`;
  const todoList = document.getElementById("todo-list");
  todoList.append(todoHTML);
}

/** Handle submitting form: gets todo and displays in list. */

function handleSubmit(evt) {
  evt.preventDefault();
  const todoId = document.getElementById("todo-id").value;
  const foundTodo = getTodoById(todoId);
  displayTodo(foundTodo);
}
```

5. **Guards:** It's common to have an *if* condition that "guards" a function (if the condition isn't true, you raise an error or return an error message or such).

   Rather than having an *if/else* around the entire body of the function (thereby indenting everything and forcing the reader to remember where they are in a long *if/else*), it is often better to handle the "guarded" case first, and then not need to have the rest of the function in an *else* block.

```
// AVOID:
if (....)  {
  return ....
} else {
  ...
  ...
  ...
  return ...
}

// RIGHT:
if (....)  {
  return ....
}
...
...
...
return ...
```

6. **Favor throwing errors** instead of returning a value when input is invalid or something goes wrong:

```
// WRONG: returning a value
function doubleValues(nums) {
  if (!Array.isArray(nums)) {
    return "Input must be an array";
  }
  // ...
}

// WRONG: not throwing an instance of an error class
function doubleValues(nums) {
  if (!Array.isArray(nums)) {
    throw "Input must be an array";
  }
  // ...
}

// RIGHT: throw an instance of an error class
function doubleValues(nums) {
  if (!Array.isArray(nums)) {
    throw new Error("Input must be an array");
  }
  // ...
}
```

7. **Arrow functions should be used only for short callbacks** (*exception*: they are also used when you need their *"this-free"* nature):

```
// RIGHT:
[1,2,3,4].map(num => num * 2);

// AVOID:
const filterOddNumbers = numbers => numbers.filter(num => num % 2 !== 0);

// RIGHT:
function filterEvenNumbers(numbers) {
  return numbers.filter(num => num % 2 === 0);
}
```

8. Prefer **named functions** over inline anonymous callbacks:

```
const signUpForm = document.getElementById("sign-up");

// WRONG: adding an anonymous callback:
signUpForm.addEventListener("submit", function(evt) {
  evt.preventDefault();
  // ...
});


// RIGHT: adding a named function:
function handleSubmit(evt) {
  evt.preventDefault();
  // ...
});

signUpForm.addEventListener("submit", handleSubmit);
```

9. Favor prefixing functions with an **underscore for "internal" functions**:

```
 class Voter {
   /** Cast vote; need to pass a validation verificationCode in.
     Returns true/false for whether vote was counted. */
   vote(verificationCode) {
     // ...
     this._validateVote()
     // ...
  }

  // Only use in the vote() method
  _validateVote() {
    ...
  }
}
```

Conditionals

1. **Short conditions:** *if* statements can be on one line, if they're short and simple:

```
if (age < 21) console.warn("Too young");

// BUT NEVER THIS:
if (age < 21)
  console.warn("Too young");
```

2. **If/Else:** *if/else* statements must always be on multiples lines with code blocks:

```
if (age < 21) {
  console.warn("Too young");
} else {
  console.log(42);
}
```

3. Think carefully about the best structure for **nested conditionals**:

```
// MEH

function isUserInstructor (user) {
  if (user.employer === 'Rithm') {
    if (user.title === 'Instructor') {
     return true;
    }
    else {
      return false;
    }
  }
  else {
    return false;
  }
}

// BETTER: use multiple if statements

function isUserInstructor (user) {
  if (user.employer !== 'Rithm') return false;
  if (user.title !== 'Instructor') return false;
  return true;
}

// BEST: use logical operators to return expressions

function isUserInstructor (user) {
  return user.employer === 'Rithm' && user.title === 'Instructor';
}
```

Ternary Operator (`? :`)

1. **Use ternaries sparingly.** They should not be nested, nor should the true/false conditions set variables. All ternaries should do something with the result of the ternary (return it from the function, pass it to a function, or set it to a variable):

```
let canVote = (age >= 18) ? "ok" : "nope";

console.log("can vote", (age >= 18) ? "ok" : "nope");

return (age >= 18) ? "ok" : "nope";

// WRONG:
(age >= 18) ? canVote = "ok" : canVote = "nope";

// WRONG: write this as an `if`/`else`:
(age >= 18) ? i++ : j++;
```

2. **Formatting ternaries:** Ternaries that can't fit on one line should have line breaks before the `?` and `:`, and those should be indented:

```
let vote = (age >= 18)
  ? getVote() * weightVote(age, height, tshirtColor)
  : 0;
```

Logging

1. **Use different "levels" of logging statements**, so you can see important messages and can filter the levels using the Chrome console:

- *console.log()*: general messages
- *console.debug()*: quieter messages that you can hide
- *console.warn()*: for warnings; appears in yellow box
- *console.error()*: to highlight something that shouldn't happen; appears in red box

2. Console messages should **provide context** for what is being logged:

```
function drawLine(x, y) {
  console.debug("drawLine", "x=", x, "y=", y);
  // ... snip ...
}
```

It's a good idea to put a console message at the top of each function, showing what is was passed as arguments. This can help find many bugs and help you visualize how the parts of the program work together.