Clean Code

Single character variable names should only be used in the context of a loop or a callback function (functions that are called back; passed in to another function to be called back later).

camelCase is the convention in JavaScript.

Try to be consistent. For instance:

```
// Consistent naming
function getPlayerScore();
function getPlayerName();
function getPlayerTag();
```

instead of:

```
// Inconsistent naming
function getUserScore();
function fetchPlayerName();
function retrievePlayer1Tag();
```

While they all mean a similar thing, at a glance you might assume different verbs were used for a specific reason (ex. "getting" might not be *quite* the same thing as "fetching" in some contexts). Also, what is the difference between `User`, `Player`, and `Player1`? If there is no difference, just use the same name, ex. `Player`.

Variables should preferably begin with a noun or an adjective (that is, a noun phrase), since they represent "things", whether that thing is a string, a number, etc. Functions represent actions so ideally begin with a verb. So:

```
// Preferable
const numberOfThings = 10;
const myName = "Thor";
const selected = true;

// Not preferable (these start with verbs, could be confused for functions)
const getCount = 10;
const showNorseGods = ["Odin", "Thor", "Loki"];
```

```
// Preferable
function getCount() {
  return numberOfThings;
}

// Not preferable (myName doesn't represent some kind of action)
function myName() {
  return "Thor";
}
```

Magic numbers can be a problem. For instance:

```
setTimeout(stopTimer, 3600000);
```

What does 3600000 mean? Even if you know that JavaScript understands time in milliseconds, you might need a calculator to figure out how many secomds or minutes it represents.

This is more descriptive:

```
const ONE_HOUR = 3600000; // Can even write as 60 * 60 * 1000;

setTimeout(stopTimer, ONE_HOUR);
```

We know that milliseconds in an hour will never change, so the all caps font is appropriate here.

Generally you should break lines that are longer than 80 characters. When manually breaking lines, you should try to break immediately *after* an operator or comma.

There are a few ways to format continuation lines. For example:

```
// This line is a bit too long
let reallyReallyLongLine = something + somethingElse + anotherThing +
howManyTacos + oneMoreReallyLongThing;

// You could format it like this
let reallyReallyLongLine =
  something +
  somethingElse +
  anotherThing +
  howManyTacos +
  oneMoreReallyLongThing;

// Or maybe like this
let anotherReallyReallyLongLine = something + somethingElse + anotherThing +
                                  howManyTacos + oneMoreReallyLongThing;
```

Semicolons are *mostly* optional in JavaScript because the JavaScript interpreter will automatically insert them if they are omitted. This functionality *can* break in certain situations, leading to bugs, so we'd recommend getting used to adding semicolons.

Don't use comments to replace git commit messages. It is not as clean and will add bloat to your files.

By using git, all this information will be neatly organized in the repository and readily accessible with `git log`.

The same applies to code that is no longer used. If you need it again in the future, just turn to your git commits. Commenting out something while testing something else is, of course, ok, but once a piece of code is not needed, just delete it.

Tell why not how. Ideally, comments do not provide pseudocode that duplicates your code. Good comments explain the *reasons* behind a piece of code. Sometimes you won't even need a comment at all!

Say we had a string where part of the text was inside square brackets and we wanted to extract the text within those brackets:

```
// Function to extract text
function extractText(s) {
  // Return the string starting after the "[" and ending at "]"
  return s.substring(s.indexOf("[") + 1, s.indexOf("]"));
}
```

The comments just describe what we can tell from the code itself. Slightly more useful comments could explain the reasons behind the code.

```
// Extracts text inside square brackets (excluding the brackets)
function extractText(s) {
  return s.substring(s.indexOf("[") + 1, s.indexOf("]"));
}
```

But often, we can make the code speak for itself without comments:

```
function extractTextWithinBrackets(text) {
  const bracketTextStart = text.indexOf("[") + 1;
  const bracketTextEnd = text.indexOf("]");
  return text.substring(bracketTextStart, bracketTextEnd);
}
```

This doesn't mean good code should lack comments. For example:

```
function calculateBMI(height, weight) {
  // The formula for BMI is weight in kilograms divided by height in meters
squared
  const heightInMeters = height / 100;
  const bmi = weight / (heightInMeters * heightInMeters);
  return bmi;
}
```

Here we help to refresh the reader on how BMI is calculated in plain English, helping the reader to see why the height needs to be converted and what the following calculation is doing. We are almost there with the naming, but the comments still adds further clarity.

In many situations, well-placed comments are priceless. They might explain why an unintuitive piece of code is necessary, or perhaps the bigger picture of why a certain function is *particularly* important to be called here and not there.

Assignment

10 Principles For Keeping Your Programming Code Clean

1. Revise your logic before coding
2. Clearly expose the structure of the page
3. Use the correct indentation
4. Write explanatory comments
5. Avoid abusing comments
6. Avoid extremely large functions
7. Use Naming Standards for functions and variables
8. Treat changes with caution
9. Avoid indiscriminate mixing of coding languages

10. Summarize your imports

Code tells you how, comments tell you why.

Even self-documenting code cannot replace comments entirely.

You should always write your code as if comments didn't exist.