Código limpio

Curso de Fundamentos

Introducción

Se podría pensar que la mayor parte del trabajo de un desarrollador consiste en escribir código. Sin embargo, en realidad, se dedica una cantidad significativa de tiempo a *leer* código. Esto incluye código escrito por otros miembros del equipo, código escrito por personas que ya no forman parte de tu equipo e incluso código que escribiste hace dos semanas pero del que quizás no recuerdes mucho.

No pienses en estos principios como algo que debes dominar de inmediato. Todo el mundo escribe código desordenado a veces, incluso los profesionales. Lo que queremos hacer aquí es darte algunas pautas que pueden ayudarte a mejorar la legibilidad de tu código a medida que avanzas. Cuanto más escribas código, mejor será, tanto en términos de legibilidad como de otros aspectos.

Pon a prueba estas ideas e intenta incorporarlas a tu proceso de pensamiento mientras escribes código, pero no te castigues por no escribir un código elegante y claro como el cristal. Concéntrate en la mejora gradual, no en la perfección.

Ahora que ya aclaramos esto, ¡comencemos!

Resumen de la lección

Esta sección contiene una descripción general de los temas que aprenderá en esta lección.

- Sepa cómo distinguir el código difícil de leer del código fácil de leer.
- Utilice principios de programación para hacer su código más limpio.
- Escribe buenos comentarios.

¿Qué es el código limpio?

Consideremos los siguientes ejemplos:

Ejemplo A:

```
const x= function (z){
   const w = "Hello ";
return w +  z


 }

x("John");
```

Ejemplo B:

```
const generateUserGreeting = function (name) {
  const greeting = "Hello ";
  return greeting + name;
};

generateUserGreeting("John");
```

¿Cuál de estos ejemplos te parece más fácil de leer? Es evidente de inmediato que el último es más significativo. Sorprendentemente, ambas funciones realizan exactamente la misma tarea (¡de la misma manera!) y ambas son código válido. Pero el segundo es mucho más legible. ¿Por qué?

En el primer ejemplo, se utilizan variables de una sola letra y la sangría y el espaciado son inconsistentes. El resultado es un fragmento de código confuso y desordenado.

Imagina que estás colaborando en un proyecto con alguien que ha escrito la primera función. ¿Cuánto tiempo te llevará descifrar lo que está pasando para poder continuar con tu trabajo? O quizás lo hayas escrito tú mismo hace algún tiempo y hayas olvidado por completo que existía. En ambas situaciones, terminarás entendiendo lo que está pasando, pero no será divertido.

El ejemplo B representa un código más limpio. Si bien es posible que no sepa qué hace cada parte, es mucho más fácil adivinar qué sucede porque las funciones y las variables tienen nombres claros. La sangría y el espaciado siguen un patrón coherente y lógico.

Se pueden usar caracteres individuales como nombres de variables en el contexto de un bucle o una función de devolución de llamada, pero evítelos en otros lugares.

Acerca de camelCase

camelCase es una convención de nombres que permite escribir varias palabras juntas sin espacios ni puntuación. En camelCase, cuando un nombre de variable consta de varias palabras como en nuestro `setTimeout` ejemplo, la primera palabra se escribe completamente en minúsculas, mientras que la primera letra de la segunda palabra (y las palabras posteriores) se escriben en mayúsculas.

Throughout this lesson, most of our variables and functions will be named using camelCase. While not every language uses this convention, it's very common in JavaScript so it'll be a good example to follow.

> #### Conventions are only conventions
>
> While this lesson shares some examples on ways to clean up code, in reality, every organization will have different specific approaches, some of which may differ slightly from our examples in this lesson. Nothing is absolute.
>
> What matters most is that these approaches all serve the same overall purpose - improve code readability and maintainability. Until a time comes where you need to follow a specific set of conventions, it is sensible to follow some convention and be consistent with them.

Naming functions and variables

In our first example, we already touched on the importance of naming things *meaningfully*. Let's break down further what makes a good variable or function name.

A good name is descriptive

In our good example, we have a variable `greeting`, to which the parameter `name` is concatenated. The function is named `generateUserGreeting` and the function does what the name suggests. Nice, clean, and understandable.

Now, try picturing a conversation with someone about the bad example. The function is named `x` with variables like `z`, and `w`. Oof, not nice.

Use consistent vocabulary

Variables of the same type ideally follow a consistent naming system. Consider the following examples from a game:

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

They all follow the same naming system of "get a thing". Now consider the following:

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

In the inconsistent example, three different verbs are used for the functions. While they all mean a similar thing, at a glance you might assume different verbs were used for a specific reason (e.g. "getting" might not be *quite* the same thing as "fetching" in some contexts). Additionally, what's the difference between `User`, `Player` and `Player1`? If there is no difference then ideally, you'd use the same name e.g. `Player`. Consistency allows for predictability.

Variables should preferably begin with a noun or an adjective (that is, a noun phrase), as they typically represent "things", whether that thing is a string, a number etc.

Functions represent actions so ideally begin with a verb.

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

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

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

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

Use searchable and immediately understandable names

Sometimes, it can be tempting to use “magic values” i.e. explicit values, such as bare numbers or strings. Let’s take another look at an example:

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

The problem is obvious. What does the magic number `3600000` mean, and how long is this timeout going to count down before executing `stopTimer`? Even if you know that JavaScript understands time in milliseconds, you’d probably need a calculator or Google to figure out how many seconds or minutes it represents.

Now, let’s make this code more meaningful by introducing a descriptive variable:

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

setTimeout(stopTimer, ONE_HOUR);
```

Much better, isn't it? The variable is declared with a descriptive name, and you don't need to perform any calculations when reading this code.

You might wonder why this variable is declared with all caps when we recommended camelCase earlier. This is a convention to be used when the programmer is absolutely sure that the variable is *truly* a constant, especially if it represents some kind of concept like a specific duration of time. We know that the milliseconds in an hour will never change, so it's appropriate here. Remember, this is only a convention. Not everyone will necessarily do things the same way.

Indentation and line length

Now it's time to head to more "controversial" topics (there's a joke about the war between coders who indent with spaces versus tabs).

What actually matters is *consistency*. Choose a way to indent and stick to it. Various JavaScript style guides recommend different options, and one is not really superior to the other. We will look at style guides and related tools in more detail later in the curriculum.

Line length

Again, different style guides will recommend different options for this one, but just about *all* of them suggest limiting the length of each line of code.

Generally, your code will be easier to read if you manually break lines that are longer than about 80 characters. Many code editors have a line in the display to show when you have crossed this threshold. When manually breaking lines, you should try to break immediately *after* an operator or comma.

Then, there are a few ways to format continuation lines. For example, you can:

```
// This line is a bit too long
let reallyReallyLongLine = something + somethingElse + another
```

```

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

// Or maybe like this
let anotherReallyReallyLongLine = something + somethingElse +
                                  howManyTacos + oneMoreReally
```

Different formats aren’t necessarily right or wrong, and different people may prefer different things. Do things in a way that makes sense to you, and stay consistent with it.

Semicolons

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 in your code, so we’d recommend getting used to adding semicolons.

Whether you do or not, again, consistency is the main thing.

About comments

Comments are a great tool but like any good tool, they can be misused. Especially for someone early in their coding journey, it might be tempting to have comments that explain *everything* the code is doing. This is generally not a good practice. Let’s look at some common pitfalls when commenting and *why* they are pitfalls.

Don’t comment when you should be using git

It might be tempting to have comments in your code that explain the changes or additions you have made. For example:

```
/**
 * 2023-01-10: Removed unnecessary code that was causing confu
 * 2023-03-05: Simplified the code (JP)
 * 2023-05-15: Removed functions that were causing bugs in pro
 * 2023-06-22: Added a new function to combine values (JR)
 */
```

The problem is that you already have a tool to track changes - git! Keeping track of these comments will become a chore, and you will have an incomplete picture of what has happened. Your files will also contain bloat that doesn't belong there.

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. Don't have something like this hanging around in your files:

```
theFunctionInUse();
// oldFunction();
// evenOlderUselessFunction();
// whyAmIEvenHereImAncient():
```

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 bracket
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);
}
```

In the first example, the comments repeat twice what the code does. But for this, you could’ve just read the code, so the comments are redundant.

In the second example, the comment clarifies the purpose of the function: extracting the text between square brackets from a string and not just “extracting text”. That’s handy, but we can do *even* better.

In the last example, no comments are needed at all. The use of descriptive functions and variable names eliminates the need for additional explanations. Pretty neat, huh?

This doesn’t mean good code should lack comments. Let’s look at an example where a comment serves a helpful purpose:

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

This comment helps 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 comment still adds further clarity.

In many situations, well-placed comments are priceless. They might explain why an unintuitive bit of code is necessary, or perhaps the bigger picture of why a certain function is *particularly* important to be called here and not there. The article linked in the assignment section goes into more depth on this.

In conclusion

Now that we've covered these ideas, it's good to return to the reminder we shared at the start. Don't try to write perfectly clean code, this will only lead to frustration. Writing "spaghetti" is inevitable; everyone does it sometimes. Just keep these ideas in mind, and with time and patience, your code will start to get cleaner.

Aprender a escribir código limpio es un proceso de mejora constante que se extenderá más allá de *completar* el Proyecto Odin. Esta lección está pensada para servir como introducción y punto de partida para ese viaje.

> *"El código excelente surge de la experiencia. La experiencia surge del código no tan bueno."*

Asignación

1. Lea 10 principios para mantener limpio su código de programación para obtener excelentes consejos para un código limpio.
2. Para ayudar a comprender mejor las buenas prácticas de comentarios, lea sobre los comentarios que nos dicen por qué funciona el código y cómo codificar sin comentarios .

Comprobación de conocimientos

Las siguientes preguntas son una oportunidad para reflexionar sobre temas clave de esta lección. Si no puede responder una pregunta, haga clic en ella para revisar el material, pero tenga en cuenta que no se espera que memorice o domine este conocimiento.

- ¿Por qué es importante escribir código limpio?
- ¿Cuáles son algunos buenos principios para mantener el código limpio?
- ¿Cuál es la diferencia entre buenos comentarios y malos comentarios?

Recursos adicionales

Esta sección contiene enlaces útiles a contenido relacionado. No es obligatoria, por lo que se la puede considerar complementaria.

- Un interesante artículo de opinión sobre el código como documentación
- Guía de estilo de Airbnb
- Métodos de encadenamiento para escribir oraciones
- Código limpio en JavaScript

Mejorar en GitHub Informar un problema Ver registro de cambios de la lección

Ver curso

Marcar como completado

Próxima lección

¡Apóyanos!

El Proyecto Odin está financiado por la comunidad. ¡Únase a nosotros para ayudar a estudiantes de todo el mundo apoyando el Proyecto Odin!

Más información

Dona ahora

THE ODIN PROJECT

Educación en codificación de alta calidad mantenida por una comunidad de código abierto.

Sobre nosotros

Acerca de

Equipo

Blog

Casos de éxito

Apoyo

Preguntas frecuentes

Contribuir

Contáctenos

Guías

Guías de la comunidad

Guías de instalación

Legal

Términos

Privacidad