Coding Best Practices

Hola!

Carlos Muiño

Fullstack Developer at
@String-Projects.

@camumino

The life of a programmer is beautiful when the code they work with is beautiful

CONTENIDO

- Contexto
- ¿Qué es una guía de estilo?
- ¿por qué usarla? Beneficios
- Puntos de una guía de estilo

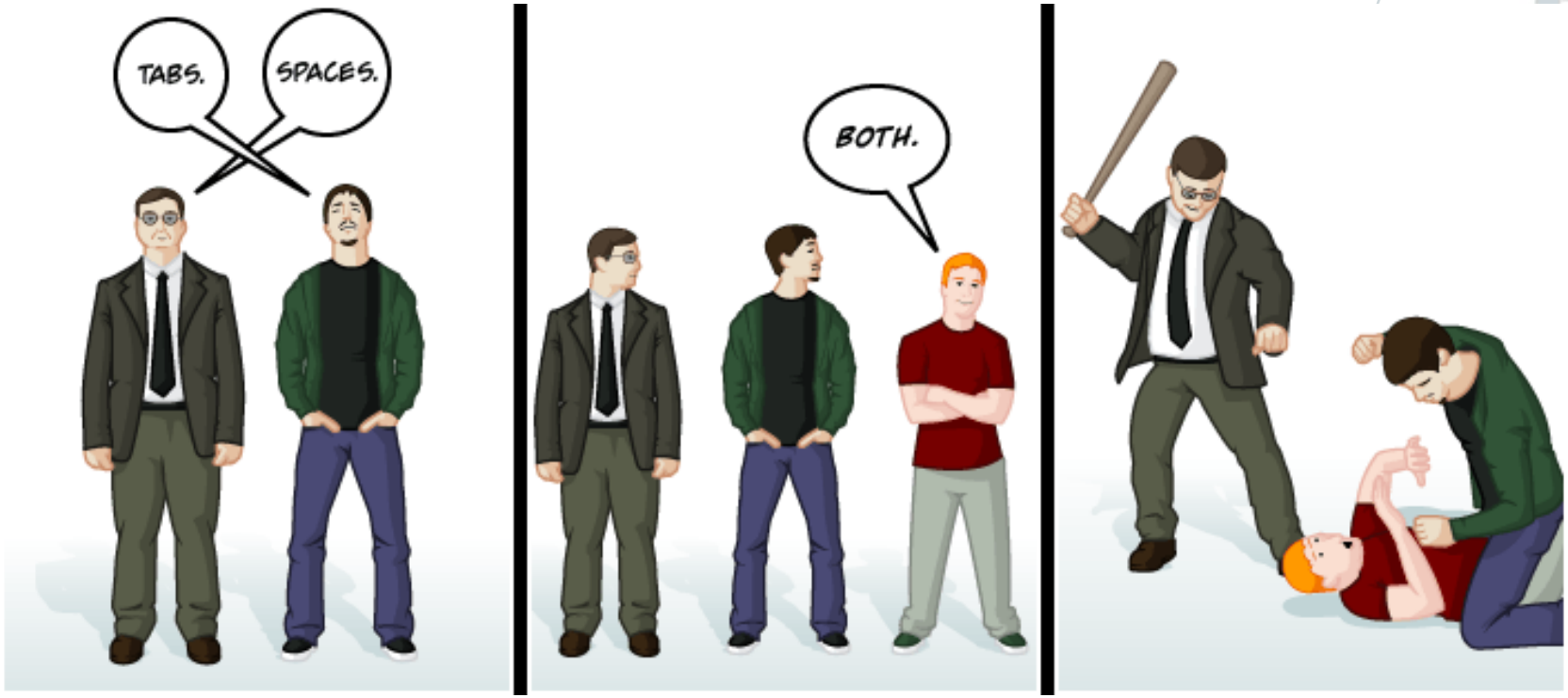
1. Contexto

Tabs vs Spaces

```
<!DOCTYPE·html>
<html·lang="en">
<head>
—<meta·charset="UTF-8">
—<title>Document</title>
</head>
<body>
—<div>
————<h1>Hello·World</h1>
—</div>
</body>
</html>
```

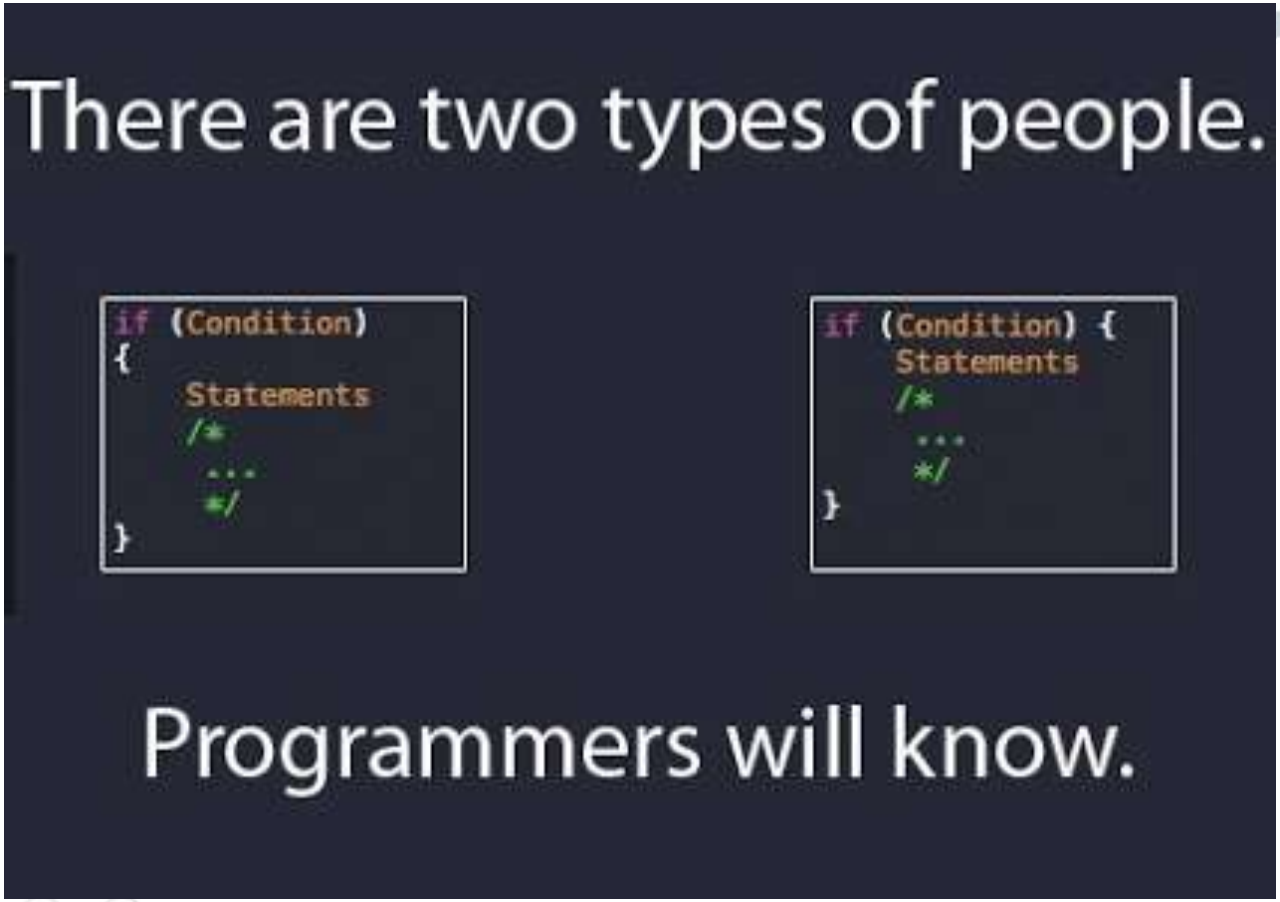
```
<!DOCTYPE·html>
<html·lang="en">
<head>
··<meta·charset="UTF-8">
··<title>Document</title>
</head>
<body>
··<div>
····<h1>Hello·World</h1>
··</div>
</body>
</html>|
```

Tabs vs Spaces



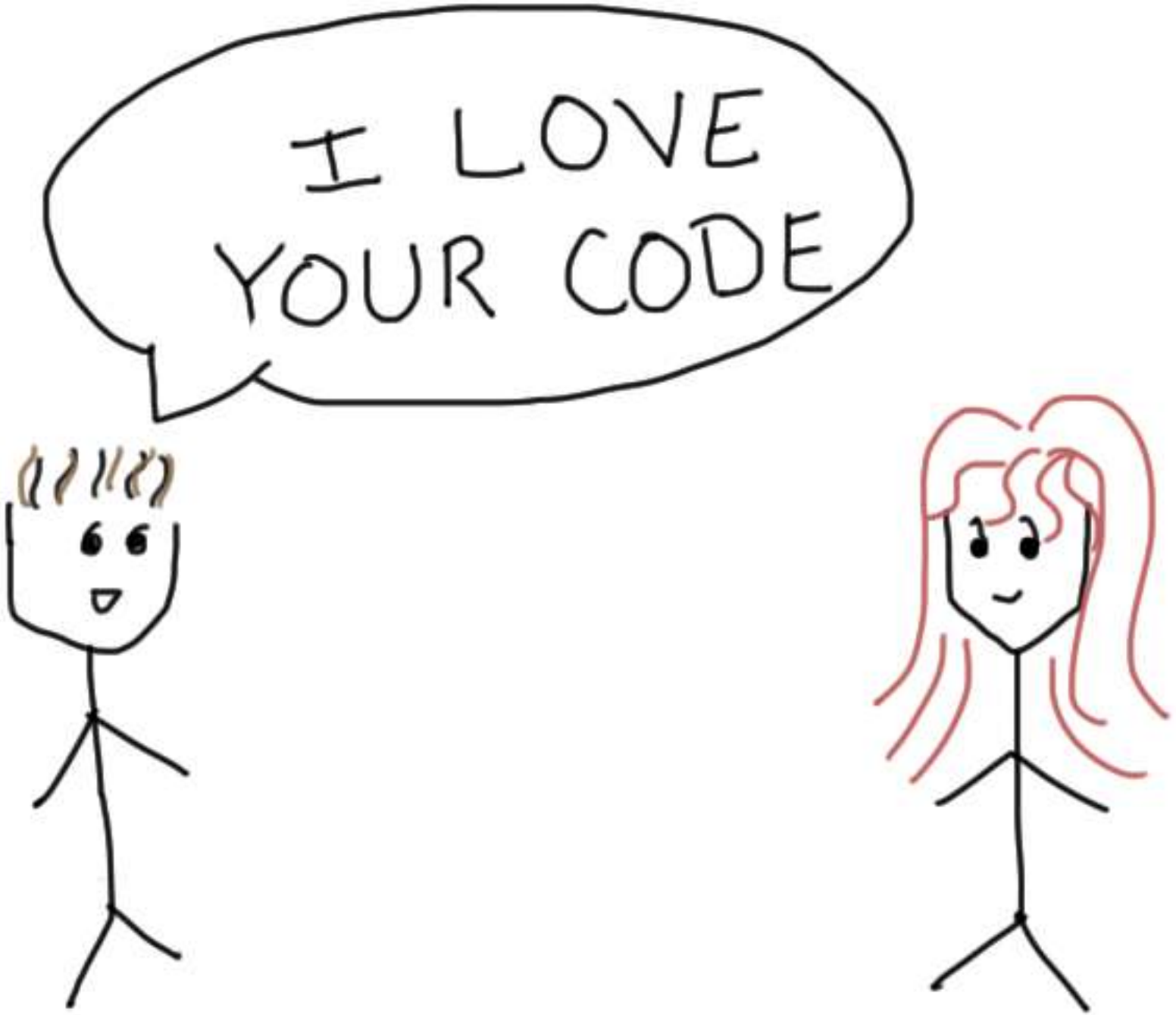
[* Debate en capitulo Silicon Valley]

Curly brace on the same line or a new line



Naming

2. ¿Qué es una guía de estilo?

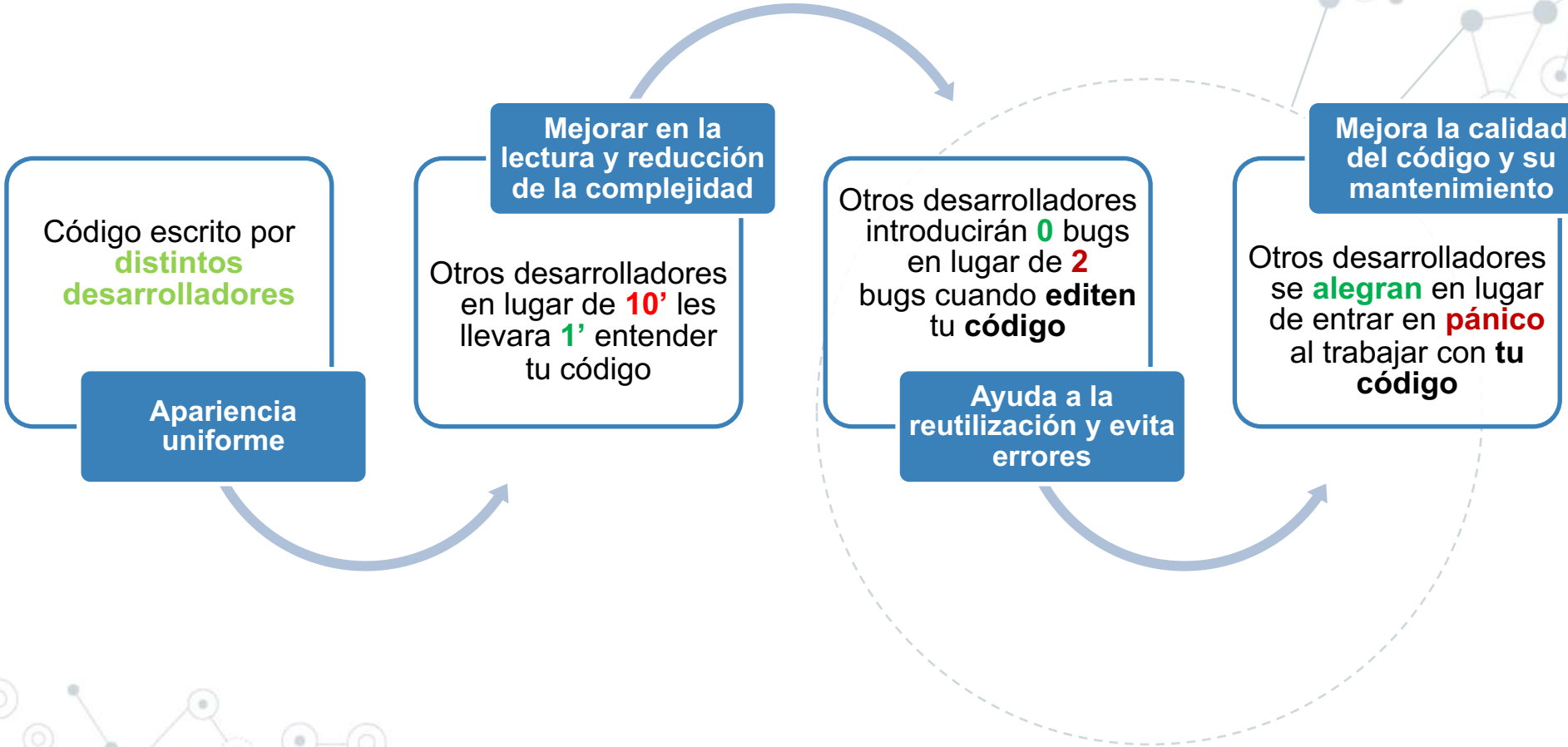

I LOVE
YOUR CODE

Definición

- Estándares de código o estilos de programación.
- Conjunto de normas usadas para escribir código fuente.
- Suma de normas, reglas y buenas practicas.
- Propias por lenguaje, compañía o incluso proyecto.

3. ¿por qué usarla? Beneficios

Comunicación – Calidad – Beneficio para todos



4. Puntos de una guía de estilo

Naming

- Nombres de variables, funciones, módulos, etc.
- Deben ser descriptivos, pronunciables y que reflejen la función que hacen o son.
- Evitar uso de dígitos en variables.
- Evitar nombre confusos o del tipo X1, Y1.

Example: A variable for taking in weight as a parameter for a truck can be named TrkWeight or TruckWeightKilograms, with TruckWeightKilograms being the preferable one

Naming (Ruby)

- Use `snake_case` for methods and variables.
- Use `CamelCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase.)
- Use `SCREAMING_SNAKE_CASE` for other constants.
- The names of predicate methods (methods that return a boolean value) should end in a question mark. (i.e. `Array#empty?`).
- The names of potentially "dangerous" methods (i.e. methods that modify `self` or the arguments, `exit!`, etc.) should end with an exclamation mark. Bang methods should only exist if a non-bang method exists. (More on this).

Naming (js)

6.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores and very rarely (when required by frameworks like Angular) dollar signs.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
errorCount          // No abbreviation.
dnsConnectionIndex  // Most people know what "DNS" stands for.
referrerUrl         // Ditto for "URL".
customerId          // "Id" is both ubiquitous and unlikely to be misunderstood.
```

Disallowed:

```
n                   // Meaningless.
nErr                // Ambiguous abbreviation.
nCompConns          // Ambiguous abbreviation.
wgcConnections      // Only your group knows what this stands for.
pcReader            // Lots of things can be abbreviated "pc".
cstmrId             // Deletes internal letters.
kSecondsPerDay      // Do not use Hungarian notation.
```

Naming (js)

6.2 Rules by identifier type

6.2.1 Package names

Package names are all `lowerCamelCase`. For example, `my.exampleCode.deepSpace`, but not `my.examplecode.deepspace` or `my.example_code.deep_space`.

6.2.2 Class names

Class, interface, record, and typedef names are written in `UpperCamelCase`. Unexported classes are simply locals: they are not marked `@private` and therefore are not named with a trailing underscore.

Type names are typically nouns or noun phrases. For example, `Request`, `ImmutableList`, or `VisibilityMode`. Additionally, interface names may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

6.2.3 Method names

Method names are written in `lowerCamelCase`. Names for `@private` methods must end with a trailing underscore.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop_`. Getter and setter methods for properties are never required, but if they are used they should be named `getFoo` (or optionally `isFoo` or `hasFoo` for booleans), or `setFoo(value)` for setters.

Underscores may also appear in JsUnit test method names to separate logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>_<expectedOutcome>`, for example `testPop_emptyStack_throws`. There is no One Correct Way to name test methods.

Naming (js)

6.2.4 Enum names

Enum names are written in `UpperCamelCase`, similar to classes, and should generally be singular nouns. Individual items within the enum are named in `CONSTANT_CASE`.

6.2.5 Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. There is no reason for a constant to be named with a trailing underscore, since private static properties can be replaced by (implicitly private) module locals.

Naming (React)

- **Extensions**: Use `.jsx` extension for React components. eslint: `react/jsx-filename-extension`
- **Filename**: Use PascalCase for filenames. E.g., `ReservationCard.jsx`.
- **Reference Naming**: Use PascalCase for React components and camelCase for their instances. eslint: `react/jsx-pascal-case`

```
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

Naming (React)

- **Component Naming**: Use the filename as the component name. For example, `ReservationCard.jsx` should have a reference name of `ReservationCard`. However, for root components of a directory, use `index.jsx` as the filename and use the directory name as the component name:

```
// bad
import Footer from './Footer/Footer';

// bad
import Footer from './Footer/index';

// good
import Footer from './Footer';
```

Naming (React)

- **Higher-order Component Naming**: Use a composite of the higher-order component's name and the passed-in component's name as the `displayName` on the generated component. For example, the higher-order component `withFoo()`, when passed a component `Bar` should produce a component with a `displayName` of `withFoo(Bar)`.

> Why? A component's `displayName` may be used by developer tools or in error messages, and having a value that clearly expresses this relationship helps people understand what is happening.

```
// bad
export default function withFoo(WrappedComponent) {
  return function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}

// good
export default function withFoo(WrappedComponent) {
  function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }

  const wrappedComponentName = WrappedComponent.displayName
    || WrappedComponent.name
    || 'Component';

  WithFoo.displayName = `withFoo(${wrappedComponentName})`;
  return WithFoo;
}
```

Naming (React)

- **Props Naming**: Avoid using DOM component prop names for different purposes.

> Why? People expect props like `style` and `className` to mean one specific thing. Varying this API for a subset of your app makes the code less readable and less maintainable, and may cause bugs.

```
// bad
<MyComponent style="fancy" />

// bad
<MyComponent className="fancy" />

// good
<MyComponent variant="fancy" />
```

Indentation – Tabs vs Spaces

- No hay mala o buena indentación.
- Únicamente debe ser consistente.
- Tabs vs Spaces.
- 2 vs 4 spaces.

Indentation – Tabs vs Spaces (ruby)

- Use soft-tabs with a two space indent.

- Indent `when` as deep as `case`.

```ruby
case
when song.name == "Misty"
  puts "Not again!"
when song.duration > 120
  puts "Too long!"
when Time.now.hour > 21
  puts "It's too late"
else
  song.play
end

kind = case year
       when 1850..1889 then "Blues"
       when 1890..1909 then "Ragtime"
       when 1910..1929 then "New Orleans Jazz"
       when 1930..1939 then "Swing"
       when 1940..1950 then "Bebop"
       else "Jazz"
       end
```

- Indent the `public`, `protected`, and `private` methods as much the method definitions they apply to. Leave one blank line above them.

```ruby
class SomeClass
  def public_method
    # ...
  end

  private
  def private_method
    # ...
  end
end
```

Indentation – Tabs vs Spaces (js)

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in 4.1.2 Nonempty blocks: K&R style).

4.2.1 Array literals: optionally "block-like"

Any array literal may optionally be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
const a = [
  0,
  1,
  2,
];

const b =
    [0, 1, 2];
```

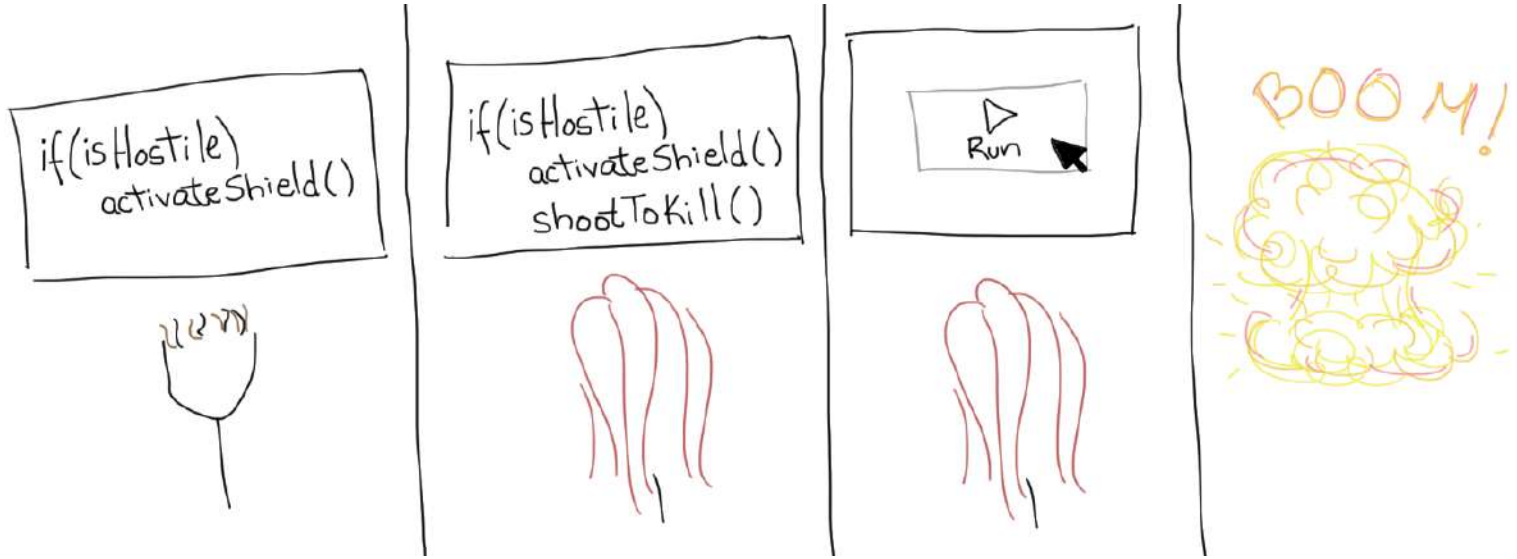
```
const c = [0, 1, 2];

someMethod(foo, [
  0, 1, 2,
], bar);
```

Other combinations are allowed, particularly when emphasizing semantic groupings between elements, but should not be used only to reduce the vertical size of larger arrays.

Indentation – Llaves

- Siempre escribir llaves en bloques de línea



Indentation – Llaves

- ¿Comenzar en una nueva línea o a continuación?
- El código a continuación ¿debe comenzar en otra nueva línea?

Indentation – Llaves (js)

4.1.1 Braces are used for all control structures

Braces are required for all control structures (i.e. `if`, `else`, `for`, `do`, `while`, as well as any others), even if the body contains only a single statement. The first statement of a non-empty block must begin on its own line.

Disallowed:

```
if (someVeryLongCondition())
  doSomething();

for (let i = 0; i < foo.length; i++) bar(foo[i]);
```

Exception: A simple if statement that can fit entirely on a single line with no wrapping (and that doesn't have an else) may be kept on a single line with no braces when it improves readability. This is the only case in which a control structure may omit braces and newlines.

```
if (shortCondition()) foo();
```

Indentation – Llaves (js)

4.1.2 Nonempty blocks: K&R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a function or class statement, or a class method. Specifically, there is *no* line break after the brace if it is followed by `else`, `catch`, `while`, or a comma, semicolon, or right-parenthesis.

Example:

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo) {
    if (condition(foo)) {
      try {
        // Note: this might fail.
        something();
      } catch (err) {
        recover();
      }
    }
  }
}
```

Indentation – Llaves (js)

4.1.3 Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters, space, or line break in between (i.e. `{}`), **unless** it is a part of a *multi-block statement* (one that directly contains multiple blocks: `if` / `else` or `try` / `catch` / `finally`).

Example:

```
function doNothing() {}
```

Disallowed:

```
if (condition) {
  // …
} else if (otherCondition) {} else {
  // …
}

try {
  // …
} catch (e) {}
```

Indentation – Llaves (react)

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// if props fit in one line then keep it on the same line
<Foo bar="bar" />

// children get indented normally
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>
```

Indentation – Llaves (react)

```
// bad
{showButton &&
  <Button />
}

// bad
{
  showButton &&
    <Button />
}

// good
{showButton && (
  <Button />
)}

// good
{showButton && <Button />}
```

Indentation – Espaciado

- Mayoría de lenguajes ignoran los espacios en blanco, sin embargo, mejora la legibilidad y comprensión.
- Los parámetros de las funciones deben estar separados por un espacio después de cada coma.
- Entre operadores.
- Después de punto y coma en bucles for.
- Antes y después del operador de asignación.

Indentation – Espaciado (ruby)

- Never leave trailing whitespace.
- Use spaces around operators, after commas, colons and semicolons, around `{` and before `}`.

```ruby
sum = 1 + 2
a, b = 1, 2
1 > 2 ? true : false; puts "Hi"
[1, 2, 3].each { |e| puts e }
```

- No spaces after `(`, `[` or before `]`, `)`.

```ruby
some(arg).other
[1, 2, 3].length
```

- No spaces after `!`.

```ruby
!array.include?(element)
```

Indentation – Espaciado (js)

4.1.3 Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters, space, or line break in between (i.e. `{}`), **unless** it is a part of a *multi-block statement* (one that directly contains multiple blocks: `if` / `else` or `try` / `catch` / `finally`).

Example:

```
function doNothing() {}
```

Disallowed:

```
if (condition) {
  // …
} else if (otherCondition) {} else {
  // …
}

try {
  // …
} catch (e) {}
```

Indentation – Espaciado (react)

- Always include a single space in your self-closing tag. eslint: `no-multi-spaces`, `react/jsx-tag-spacing`

```
// bad
<Foo/>

// very bad
<Foo                 />

// bad
<Foo
 />

// good
<Foo />
```

- Do not pad JSX curly braces with spaces. eslint: `react/jsx-curly-spacing`

```
// bad
<Foo bar={ baz } />

// good
<Foo bar={baz} />
```

Indentation – Longitud y salto de línea

- Largas líneas son complicadas de leer.
- Evitar escribir largas líneas horizontales.
- Limitar longitud de línea alrededor de 80 caracteres.

Indentation – Longitud y salto de línea (js)

4.4 Column limit: 80

JavaScript code has a column limit of 80 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in 4.5 Line-wrapping.

Exceptions:

1. `goog.module`, `goog.require` and `goog.requireType` statements (see 3.3 goog.module statement and 3.6 goog.require and goog.requireType statements).
2. ES module `import` and `export from` statements (see 3.4.1 Imports and 3.4.2.4 export from).
3. Lines where obeying the column limit is not possible or would hinder discoverability. Examples include:
   - A long URL which should be clickable in source.
   - A shell command intended to be copied-and-pasted.
   - A long string literal which may need to be copied or searched for wholly (e.g., a long file path).

Indentation – Longitud y salto de línea (js)

4.5 Line-wrapping

Terminology Note: *Line wrapping* is breaking a chunk of code into multiple lines to obey column limit, where the chunk could otherwise legally fit in a single line.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

> Note: While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit may be line-wrapped at the author's discretion.

> Tip: Extracting a method or local variable may solve the problem without the need to line-wrap.

Indentation – Longitud y salto de línea (js)

4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**.

Preferred:

```
currentEstimate =
    calc(currentEstimate + x * currentEstimate) /
        2.0;
```

Discouraged:

```
currentEstimate = calc(currentEstimate + x *
    currentEstimate) / 2.0;
```

In the preceding example, the syntactic levels from highest to lowest are as follows: assignment, division, function call, parameters, number constant.

Operators are wrapped as follows:

1. When a line is broken at an operator the break comes after the symbol. (Note that this is not the same practice used in Google style for Java.)
   a. This does not apply to the "dot" (`.`), which is not actually an operator.
2. A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
3. A comma (`,`) stays attached to the token that precedes it.

> Note: The primary goal for line wrapping is to have clear code, not necessarily code that fits in the smallest number of lines.

Comentarios

- Código documentado para un mejor entendimiento.
- Definir su función de forma clara.
- Proveen información adicional que no es legible en el código mismo.
- No deben explicar paso a paso lo que el código hace (eso lo debe hacer tu excelente nombrado de función/variable).

Tus comentarios deben decir el PORQUE algo ha sido programado así, el código ya te dice el COMO

Cuando comentar

Si tu mismo te encuentras pensando algo del tipo: implementé esto de esta forma debido a tal decisión técnica y/o de negocio, entonces debes comentarlo

Comentarios

- Evitar comentar cosas claras o irrelevantes.

```php
// get the country code
$country_code = get_country_code($_SERVER['REMOTE_ADDR']);

// if country code is US
if ($country_code == 'US') {

    // display the form input for state
    echo form_input_state();
}
```

- Comentar en exceso provocara que nuestros colegas se “cansen” de leer comentarios y no se percataran de los comentarios realmente relevantes, pudiéndose así provocar bugs, etc

TODOs

- Comentario para “guardar” notas en el código.
- Trackear problemas.
- Marcar deuda técnica.
- Dejarte notas para volver al foco por si tenemos que cambiar a otro desarrollo.
- Estructura uniforme: TODO[nombre]: explicación/nota

Inglés

- Nombres en ingles mas cortos -> Mejora en lectura
- Facilidad para preguntar en foros pudiendo copiar y pegar directamente el código
- Integración en equipos internacionales

Código simple - Nesting

```php
function do_stuff() {

// ...

    if (is_writable($folder)) {

        if ($fp = fopen($file_path,'w')) {

            if ($stuff = get_some_stuff()) {

                if (fwrite($fp,$stuff)) {

                    // ...

                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

```php
function do_stuff() {

// ...

    if (!is_writable($folder)) {
        return false;
    }

    if (!$fp = fopen($file_path,'w')) {
        return false;
    }

    if (!$stuff = get_some_stuff()) {
        return false;
    }

    if (fwrite($fp,$stuff)) {
        // ...
    } else {
        return false;
    }
}
```

Código simple - Legibilidad

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

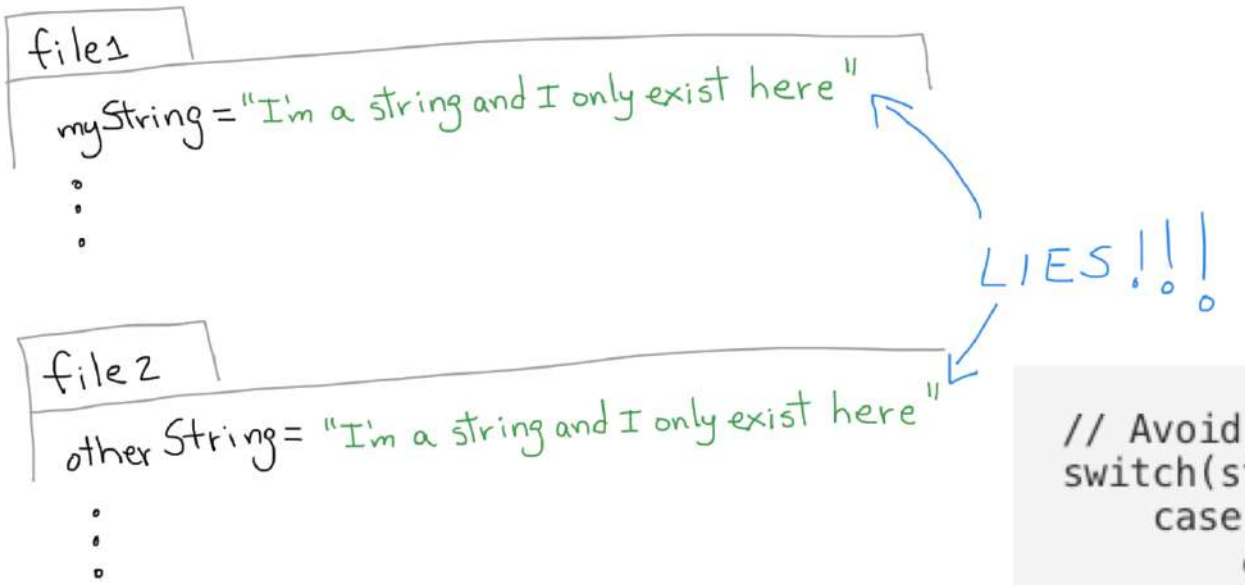
```
if (hours < 24 && minutes < 60 && seconds < 60)
    return true;
else
    return false;
```

```
return hours < 24 && minutes < 60 && seconds < 60;
```

Código simple – Funciones complejas

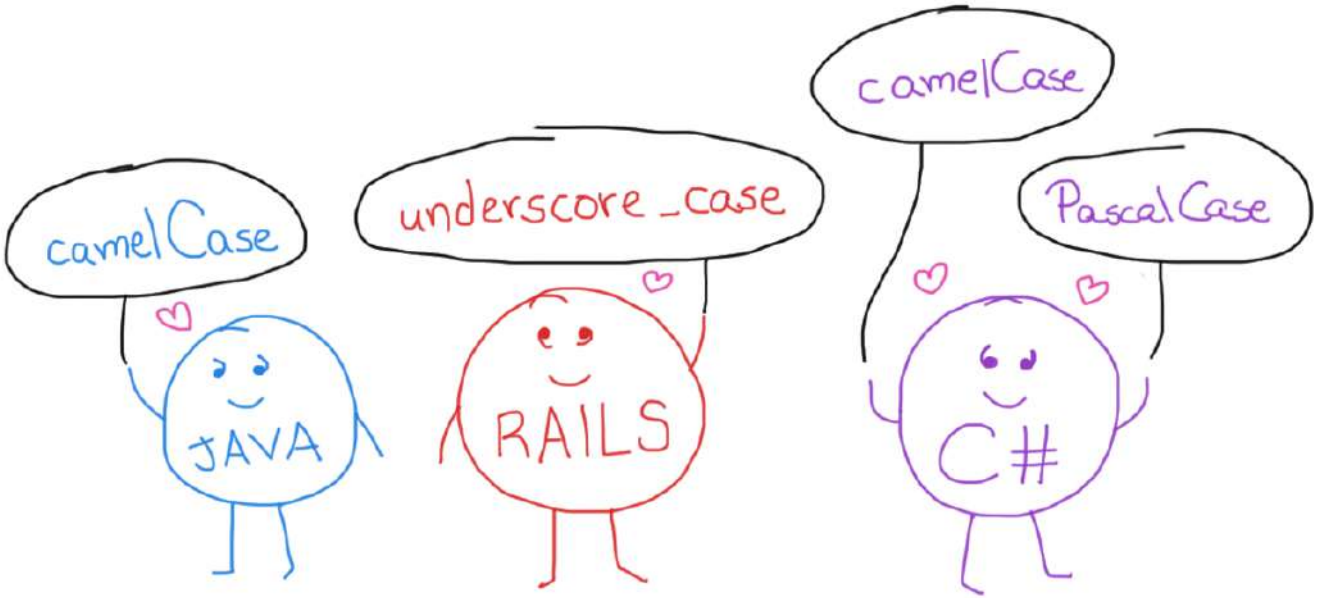
- La función debe verse en tu pantalla sin hacer scroll
- Que haga lo que “Prometio” su nombre, ni mas ni menos.
- El menor numero de argumentos posible
- Si hay muchos argumentos, debes pensar en separar en distintas funciones
- Cuanto menor código, un cambio supondrá menos impacto en el código en general

Strings Hardcodeados o números mágicos




```
// Avoid this
switch(status) {
    case 1:
        do_this();
        break;
    case 2:
        do_that();
        break;
    default:
        do_something_else();
}
```

Convenios



[Github Ruby Style Guide]
[Google JavaScript Style Guide]
[Airbnb React/JSX Style Guide]

Gracias!

¿Preguntas?

@camumino

carlos.muino@string-projects.com