|  |  |
| --- | --- |
| Denominación | Taller estándares de codificación |
| Nombre y código del Programa de formación: | TO Análisis y Desarrollo Software |
| Competencia: | Desarrollar La Solución De Software De Acuerdo Con El Diseño Y Metodologías De Desarrollo |
| Identificación de la actividad: | Definir estándares de codificación |

**Taller estándares de codificación**

**Instrucciones:** Leer con atención la actividad propuesta por el instructor, desarrollar cada uno de los puntos propuestos y presentarla de forma digital.

1. Realizar la lectura del libro “Clean JavaScript” de la página 18 a la 31 y responder las siguientes preguntas

* Que es clean code

Para mi comrende código **fácil de leer, mantener y extender**, escrito con estándares basados en: **Claridad**, **Simplicidad**, **Colaboración** (que otros lo entiendan), **Eficiencia** sin sacrificar legibilidad.

Como dicen los expertos, es código que **"parece escrito por alguien que se preocupa** y **no tiene sorpresas"**.

* Nombrar 20 tips que se deben poner en práctica para manejar un código limpio

\*Los arrays son una lista iterable de elementos, generalmente del mismo tipo. Es por

ello que pluralizar el nombre de la variable puede ser una buena idea:

Variables, nombres y ámbito

Arrays

1 //bad

2 const fruit = ['manzana', 'platano', 'fresa'];

3 // regular

4 const fruitList = ['manzana', 'platano', 'fresa'];

5 // good

6 const fruits = ['manzana', 'platano', 'fresa'];

7 // better

8 const fruitNames = ['manzana', 'platano', 'fresa'];

\* Los booleanos solo pueden tener 2 valores: verdadero o falso. Dado esto, el uso de

prefijos como is, has y can ayudará inferir el tipo de variable, mejorando así la

legibilidad de nuestro código.

Booleanos

1 //bad

2 const open = true;

3 const write = true;

4 const fruit = true;

6 // good

7 const isOpen = true;

8 const canWrite = true;

9 const hasFruit = true;

\* Para los números es interesante escoger palabras que describan números, como min,

max o total:

Variables, nombres y ámbito

Números

1 //bad

2 const fruits = 3;

4 //better

5 const maxFruits = 5;

6 const minFruits = 1;

7 const totalFruits = 3;

\* Los nombres de las funciones deben representar acciones, por lo que deben construirse

usando el verbo que representa la acción seguido de un sustantivo. Estos deben de

ser descriptivos y, a su vez, concisos. Esto quiere decir que el nombre de la función

debe expresar lo que hace, pero también debe de abstraerse de la implementación de

la función.

Funciones

1 //bad

2 createUserIfNotExists()

3 updateUserIfNotEmpty()

4 sendEmailIfFieldsValid()

6 //better

7 createUser(...)

8 updateUser(...)

9 sendEmail()

En el caso de las funciones de acceso, modificación o predicado, el nombre debe ser

el prefijo get, set e is, respectivamente.

Variables, nombres y ámbito 20

Funciones de acceso, modificación o predicado

1 getUser()

2 setUser(...)

3 isValidUser()

\* Get y set

En el caso de los getters y setters, sería interesante hacer uso de las palabras clave

get y set cuando estamos accediendo a propiedades de objetos. Estas se introdujeron

en ES6 y nos permiten definir métodos accesores:

Get y set

1 class Person {

2 constructor(name) {

3 this.\_name = name;

4 }

6 get name() {

7 return this.\_name;

8 }

10 set name(newName) {

11 this.\_name = newName;

12 }

13 }

14

15 let person = new Person(‘Miguel’);

16 console.log(person.name); // Outputs ‘Miguel

\* Las clases y los objetos deben tener nombres formados por un sustantivo o frases de

sustantivo como User, UserProfile, Account o AdressParser. Debemos evitar nombres

genéricos como Manager, Processor, Data o Info.

Variables, nombres y ámbito 21

Hay que ser cuidadosos a la hora de escoger estos nombres, ya que son el paso previo

a la hora de definir la responsabilidad de la clase. Si escogemos nombres demasiado

genéricos tendemos a crear clases con múltiples responsabilidades.

\* Además de escribir nombres adecuados para las variables, es fundamental entender

cómo funciona su *scope* en JavaScript. El *scope*, que se traduce como “ámbito” o

“alcance” en español, hace referencia a la visibilidad y a la vida útil de una variable.

El ámbito, en esencia, determina.

\* Cualquier variable que no esté dentro de un bloque de una función, estará dentro

del ámbito global. Dichas variables serán accesibles desde cualquier parte de la

aplicación:

**Ámbito global**

1 **let** greeting = ‘hello world!’;

2

3 **function** greet(){

4 console.log(greeting);

5 }

6

7 greet(); *//”Hello world”;*

*\** *Como comentamos en el apartado de “Uso correcto de var, let y const”,*

*para definir variables con alcance de bloque debemos hacer uso de let o const:*

*Variables, nombres y ámbito 22*

***Ámbito de bloque***

*1 {*

*2* ***let*** *greeting = “Hello world!”;*

*3* ***var*** *lang = “English”;*

*4 console.log(greeting); //Hello world!*

*5 }*

*6*

*7 console.log(lang);//”English”*

*8 console.log(greeting);//// Uncaught ReferenceError: greeting is not def\*

*9 ined*

*En este ejemplo queda patente que las variables definidas con var se pueden emplear*

*fuera del bloque, ya que este tipo de variables no quedan encapsuladas dentro de los*

*bloques. Por este motivo, y de acuerdo con lo mencionado anteriormente, debemos*

*evitar su uso para no encontrarnos con comportamientos inesperados.*

*\** *El ámbito de las variables en JavaScript tiene un comportamiento de naturaleza*

*estática. Esto quiere decir que se determina en tiempo de compilación en lugar de*

*en tiempo de ejecución. Esto también se suele denominar* ***ámbito léxico (lexical***

***scope).*** *Veamos un ejemplo:*

***Ámbito estático vs. dinámico***

*1* ***const*** *number = 10;*

*2* ***function*** *printNumber() {*

*3 console.log(number);*

*4 }*

*5*

*6* ***function*** *app() {*

*7* ***const*** *number = 5;*

*8 printNumber();*

*9 }*

*10*

*11 app(); //10*

*Variables, nombres y ámbito 23*

*En el ejemplo, console.log (number) siempre imprimirá el número 10 sin importar*

*desde dónde se llame la función printNumber(). Si JavaScript fuera un lenguaje con*

*el ámbito dinámico, console.log(number) imprimiría un valor diferente dependiendo*

*de dónde se ejecutará la función printNumber().*

*\** *En JavaScript las declaraciones de las variables y funciones se asignan en memoria*

*en tiempo de compilación; a nivel práctico es como si el intérprete moviera dichas*

*declaraciones al principio de su ámbito. Este comportamiento es conocido como*

***hoisting****. Gracias al hoisting podríamos ejecutar una función antes de su declaración:*

***Hoisting***

*1 greet(); //”Hello world”;*

*2* ***function*** *greet(){*

*3* ***let*** *greeting = ‘Hello world!’;*

*4 console.log(greeting);*

*5 }*

*Al asignar la declaración en memoria es como si “subiera” la función al principio de*

*su ámbito:*

***Hoisting***

*1* ***function*** *greet(){*

*2* ***let*** *greeting = ‘Hello world!’;*

*3 console.log(greeting);*

*4 }*

*5 greet(); //”Hello world”;*

*En el caso de las variables, el hoisting puede generar comportamientos inesperados,*

*ya que como hemos dicho solo aplica a la declaración y no a su asignación:*

*Variables, nombres y ámbito 24*

***Hoisting***

*1* ***var*** *greet = "Hi";*

*2 (****function*** *() {*

*3 console.log(greet);// "undefined"*

*4* ***var*** *greet = "Hello";*

*5 console.log(greet); //”Hello”*

*6 })();*

*En el primer console.log del ejemplo, lo esperado es que escriba “Hi”, pero como*

*hemos comentado, el intérprete “eleva” la declaración de la variable a la parte*

*superior de su scope. Por lo tanto, el comportamiento del ejemplo anterior sería*

*equivalente a escribir el siguiente código:*

***Hoisting***

*1* ***var*** *greet = "Hi";*

*2 (****function*** *() {*

*3* ***var*** *greet;*

*4 console.log(greet);// "undefined"*

*5 greet = "Hello";*

*6 console.log(greet); //”Hello”*

*7 })();*

*He usado este ejemplo porque creo que es muy ilustrativo para explicar el concepto*

*de* ***hoisting****, pero volver a declarar una variable con el mismo nombre y además usar*

*var para definirlas es muy mala idea.*

\* Las funciones son la entidad organizativa más básica en cualquier programa. Por ello,

deben resultar sencillas de leer y de entender, además de transmitir claramente su

intención.

\* **“Sabemos que estamos desarrollando código limpio cuando cada función hace**

**exactamente lo que su nombre indica”.**

**\* “No comentes el código mal escrito, reescríbelo”.**

**\* “El buen código siempre parece estar escrito por alguien a quien le importa”.**

**\*** **STUPID es simplemente un acrónimo**

**basado en seis *code smells* que describen cómo NO debe ser el *software* que**

**desarrollamos.**

**• Singleton: patrón singleton**

**• Tight Coupling: alto acoplamiento**

**• Untestability: código no testeable**

**• Premature optimization: optimizaciones prematuras**

**• Indescriptive Naming: nombres poco descriptivos**

**• Duplication: duplicidad de código.**

**\*** **Los principios de SOLID nos indican cómo organizar nuestras funciones y estructuras**

**de datos en componentes y cómo dichos componentes deben estar interconectados.**

**Normalmente éstos suelen ser clases, aunque esto no implica que dichos principios**

**solo sean aplicables al paradigma de la orientación a objetos, ya que podríamos tener**

**simplemente una agrupación de funciones y datos, por ejemplo, en una *Closure*. En**

**definitiva, cada producto *software* tiene dichos componentes, ya sean clases o no, por**

**lo tanto tendría sentido aplicar los principios SOLID.**

• **S**ingle Responsibility: Responsabilidad única.

• **O**pen/Closed: Abierto/Cerrado.

• **L**iskov substitution: Sustitución de Liskov.

• **I**nterface segregation: Segregación de interfaz.

• **D**ependency Inversion: Inversión de dependencia.

Es importante resaltar que se trata de principios, no de reglas. Una regla es de

obligatorio cumplimiento, mientras que los principios son recomendaciones que

pueden ayudar a hacer las cosas mejor

\***Código Autodocumentado**

* ✔️ isUserActive() en lugar de checkFlag(user).

\***Tests Automatizados**

* Código testeable = código desacoplado y limpio (como decía Dave Thomas).

\* **Eliminar Código Muerto**

* Borrar funciones/comentarios obsoletos.

1. Analizar las siguientes frases de programadores celebres y debajo de cada una escribir lo que entendió de esta

* Bjarne StrousTrup decía **“Código limpio es aquel que es elegante y eficiente. Las dependencias deben ser mínimas para facilitar el mantenimiento y la gestión de errores completa”**

El código debe ser bien estructurado y optimizado y reducir acoplamientos para facilitar mantenimiento y manejo de errores.

* Grady Booch dice **“El código limpio tiene que ser simple y directo, debería leerse con la misma facilidad que un texto bien escrito. Y no ocultar nunca la intención del programador”**

Como un texto bien escrito, sin ambigüedades y el propósito del código debe ser evidente.

* Dave Thomas dice **“el código limpio es aquel que puede entenderlo otro programador. Tiene test automáticos y nombres con sentido, ofrece una api clara y concisa”**

Usar nombres significativos y APIs claras, garantizando confiabilidad y facilitar cambios.

* Michael Feathers **dice “el código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que se pueda hacer para mejorarlo”**

Demuestra atención al detalle y refleja un diseño bien pensado.

* Ward Cunnigham dice **“Sabemos que estamos trabajando con código limpio cuando cada función que leemos resulta ser lo que esperamos, sin sorpresas”**

El código debe comportarse de manera intuitiva, sin lógica oculta o efectos secundarios inesperados.

Según el video <https://www.youtube.com/watch?v=Ayh2eiW4KOU&t=497s> listar 10 claves a la hora de nombrar según el tipo de datos

1. Que es indentar y que objetivo o función tiene

Indentar en la programación significa añadir espacios en blanco (usualmente tabuladores) al principio de una línea de código para mejorar la legibilidad y organización. El objetivo principal de indentar es facilitar la lectura, comprensión y mantenimiento del código.

1. Cuando y de qué forma se debe hacer un comentario en el código

Los comentarios en el código son notas explicativas que se agregan al código para que las personas que lo lean puedan entender mejor lo que hace el mismo. Se deben usar de manera inteligente, no en exceso, y siguiendo unas buenas prácticas para que sean útiles.

1. ¿Qué hay de malo con el estilo de código a continuación? Corregir según los estándares de codificación

function pow(x,n)

{

let result=1;

for(let i=0;i<n;i++) {result\*=x;}

return result;

}

let x=prompt("x?",''), n=prompt("n?",'')

if (n<=0)

{

alert(`Power ${n} is not supported, please enter an integer number greater than zero`);

}

else

{

alert(pow(x,n))

}

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**PRODUCTO(S) ENTREGABLE(S)**

Documento digital con las respuestas y/o solución del *Taller* estándares de codificación

**FORMA DE ENTREGA**

El documento con la solución del taller conceptos de Taller estándares de codificación deberá subirse a la plataforma de formación dispuesta por el SENA.