




MejorCodigo

 2025-01   5 min.

Pregunta 1:

Para cualquier función o funcionalidad que se quiera desarrollar, hay una sola implementación que es mejor que todas?

Pregunta 2:

Que hace que una implementación de código sea mejor que otra?

Ejemplo 1

Cual es mejor? Porque?

```
1  function f(n) {  
2    let a = [];  
3    for (let i = 2; i <= n; i++) {  
4      let p = 1;  
5      for (let j = 2; j < i; j++) {  
6        if (i % j === 0) p = 0;  
7      }  
8      if (p) a.push(i);  
9    }  
10   return a;  
11 }
```

```
1  function buscarPrimos(n) {  
2    let primos = [];  
3    for (let i = 2; i <= n; i++) {  
4      let esPrimo = true;  
5      for (let j = 2; j < i; j++) {  
6        if (i % j === 0) esPrimo = false;  
7      }  
8      if (esPrimo) primos.push(i);  
9    }  
10   return primos;  
11 }
```

1. Legibilidad

Es importante que un código sea **legible**.

Sin embargo, dos desarrolladores pueden ver un mismo código y tener opiniones diferentes sobre que tan legible es el mismo, es algo subjetivo.

Ejemplo 2

Cual es mejor? Porque?

```
1  function sumarNumeros(n) {  
2    let numeros = [];  
3    for (let i = 1; i <= n; i++) {  
4      numeros.push(i);  
5    }  
6    return numeros.reduce((suma, num) => suma + num, 0);  
7  }
```

```
1  function sumarNumeros(n) {  
2    let suma = 0;  
3    for (let i = 1; i <= n; i++) {  
4      suma += i;  
5    }  
6    return suma;  
7  }
```

2. Consumo de Memoria

En lo posible, se quiere minimizar el consumo de memoria de nuestro código.

Guardar datos innecesarios o de un solo uso es una forma en que se "malgasta" la memoria.

Ejemplo 3

Cual es mejor? Porque?

```
1  function suma(n) {  
2    let suma = 0;  
3    for (let i = 1; i <= n; i++) {  
4      suma += i;  
5    }  
6    return suma;  
7  }
```

```
1  function suma(n) {  
2    return (n * (n + 1)) / 2;  
3  }
```


Ejemplo 4

Cual es mejor? Porque?

```
1  function busquedaLineal(lista, objetivo) {  
2    for (let i = 0; i < lista.length; i++) {  
3      if (lista[i] === objetivo) {  
4        return i;  
5      }  
6    }  
7    return -1;  
8  }
```

```
1  function busquedaLineal(lista, objetivo) {  
2    let indice = -1;  
3    for (let i = 0; i < lista.length; i++) {  
4      if (lista[i] === objetivo) {  
5        indice = i;  
6      }  
7    }  
8    return indice;  
9  }
```

Ejemplo 5

Cual es mejor? Porque?

```
1  function busquedaLineal(lista, objetivo) {  
2    for (let i = 0; i < lista.length; i++) {  
3      if (lista[i] === objetivo) {  
4        return i;  
5      }  
6    }  
7    return -1;  
8  }
```

```
1  function busquedaLineal(lista, objetivo) {  
2    let indice = -1;  
3    for (let i = 0; i < lista.length; i++) {  
4      if (lista[i] === objetivo) {  
5        indice = i;  
6        break;  
7      }  
8    }  
9    return indice;  
10 }
```

3. Tiempo de Procesamiento

En lo posible, se quiere minimizar el tiempo de procesamiento de nuestro código.

Reducir o remover la cantidad o longitud de ciclos es una buena forma de reducir el tiempo de procesamiento.

Ejemplo 6

Cual es mejor? Porque?

```
1  const descuentos = {
2    normal: 1,
3    vip: 0.9,
4    estudiante: 0.85,
5    mayor: 0.8,
6  };
7
8  function calcularPrecioConDescuento(precio, tipoCliente) {
9    return precio * (descuentos[tipoCliente] ?? 1);
10 }
```

```
1  function calcularPrecioConDescuento(precio, tipoCliente) {
2    if (tipoCliente === "normal") {
3      return precio;
4    } else if (tipoCliente === "vip") {
5      return precio * 0.9;
6    } else if (tipoCliente === "estudiante") {
7      return precio * 0.85;
8    } else if (tipoCliente === "mayor") {
9      return precio * 0.8;
10   }
11
12   return precio;
13 }
```

4. Mantenibilidad

En lo posible, se quiere tener y escribir código mantenible, que sea fácil de hacerle soporte, o modificar.

Funciones pequeñas y objetos "diccionarios" están usualmente relacionados a la mantenibilidad.

También la reducción de incidencia de un valor "quemado" a favor de tener el valor guardado en una variable y acceder a él donde se necesite.

Ejemplo 7

Cual es mejor? Porque?

```
1  function validarUsuario(usuario) {  
2    return usuario.nombre && usuario.edad >= 18;  
3  }  
4  
5  function procesarUsuario(usuario) {  
6    return { ...usuario, registro: new Date() };  
7  }  
8  
9  function guardarUsuarios(usuarios, db) {  
10   usuarios  
11     .filter(validarUsuario)  
12     .map(procesarUsuario)  
13     .forEach((usuario) => db.insert("usuarios", usuario));  
14 }
```

```
1  function guardarUsuarios(usuarios, db) {  
2    for (let i = 0; i < usuarios.length; i++) {  
3      if (!usuarios[i].nombre || usuarios[i].edad < 18) {  
4        continue;  
5      }  
6      usuarios[i].registro = new Date();  
7      db.insert("usuarios", usuarios[i]);  
8    }  
9  }
```

5. Escalabilidad

En lo posible, se quiere tener y escribir código escalable, que sea fácil de agregar funcionalidades nuevas.

Mantenibilidad vs. Escalabilidad

Estos dos conceptos van muy de la mano, ambos "piden" funciones pequeñas con responsabilidades claras.

Sin embargo...

- Mantenibilidad habla de modificar o mantener lo ya implementado.
- Escalabilidad habla de extender o agregar la funcionalidad de lo implementado.

Y también, hay otros tipos de escalabilidad.

Escalabilidad de código

Ya la vimos.

Escalabilidad de Uso

Desplegaste un servidor.

Que tan bien maneja ese servidor 100 conexiones? 1.000? 1.000.000? 1.000.000.000?

Escalabilidad de Tamaño de Datos

Desplegaste una API+DB.

Que tan bien maneja esa API+DB una búsqueda con filtros de una tabla con 100 filas? 1.000.000?
1.000.000.000?

Ejemplo 8

Cual es mejor? Porque?

```
1  function mensaje(nombre) {  
2    return "Hola, " + nombre + "!";  
3  }
```

```
1  const mensaje = (nombre) => `Hola, ${nombre}!`;
```

6. Compatibilidad

En lo posible, se quiere tener y escribir código compatible con todos los entornos que se estén utilizando.

Es importante tener en cuenta cuál es el entorno más viejo o más restrictivo que utilizamos para asegurarnos que nuestro código funcione correctamente.

Que hace que una implementación de código sea mejor que otra?

1. Legibilidad – Código claro y fácil de entender.
2. Consumo de memoria – Uso eficiente de recursos.
3. Procesamiento – Algoritmos y optimización.
4. Mantenibilidad – Código modular y fácil de modificar.
5. Escalabilidad – Capacidad y facilidad de manejar más casos/carga/datos.
6. Compatibilidad – Funcionamiento en diferentes entornos y versiones.

Pregunta 3:

Es posible cumplir con las 6 cosas al mismo tiempo?

Legibilidad / Memoria / Procesamiento / Mantenibilidad / Escalabilidad / Compatibilidad

Pregunta 4:

Cuales son las mas importantes? Como priorizamos?

Legibilidad / Memoria / Procesamiento / Mantenibilidad / Escalabilidad / Compatibilidad

Que hace este codigo?

Este codigo es "mejor".

```
1  function t(d) {  
2    let m = {};  
3    return (  
4      d.forEach((x) => (m[x.id] = { ...x, c: [] }))) ||  
5      d.forEach((x) => x.p && m[x.p].c.push(m[x.id])) ||  
6      d.filter((x) => !x.p).map((x) => m[x.id])  
7    );  
8  }  
9  
10 console.log(  
11   t([  
12     { id: 1, p: null, name: "A" },  
13     { id: 2, p: 1, name: "B" },  
14     { id: 3, p: 1, name: "C" },  
15     { id: 4, p: 2, name: "D" },  
16   ])  
17 );
```

Que hace este código?

```
1  function construirArbol(datos) {
2    let mapa = {};
3
4    // Paso 1: Crear un mapa de nodos por ID
5    datos.forEach((elemento) => {
6      mapa[elemento.id] = { ...elemento, hijos: [] };
7    });
8
9    // Paso 2: Construir la estructura jerárquica
10   let raiz = [];
11   datos.forEach((elemento) => {
12     if (elemento.p !== null) {
13       mapa[elemento.p].hijos.push(mapa[elemento.id]);
14     } else {
15       raiz.push(mapa[elemento.id]);
16     }
17   });
18
19   return raiz;
20 }
```


Fin