

## PRÁCTICA CALIFICADA 4

Cesar Jesus Lara Avila

Calagua Mallqui Jairo Andre

20210279F

Facultad de Ciencias, Universidad Nacional de Ingeniería

noviembre del 2023

## PREGUNTA 1:

### Implementación de un comando grep simplificado en JavaScript

Según las indicaciones dadas, debo implementar un comando grep simplificado, que admita la búsqueda de cadenas fijas. Este debe tener tres argumentos:

- La cadena a buscar.
- Cero o más indicadores para personalizar el comportamiento del comando.
- Uno o más archivos para buscar.

Además de que se nos muestra los indicadores que admite el comando grep.

Para facilitar esto, el archivo grep.js se configuró con un shebang y un comentario que explica lo que hace.

Entonces, realizo la implementación de grep en JavaScript con los pasos que se me indican:

```
JS grep.js X
C: > Users > calag > Desktop > JS grep.js > grep
1  #!/usr/bin/env node
2
3  const fs = require('fs');
4
5  const searchString = process.argv[2];
6  const flags = process.argv.slice(3, -1);
7  const files = process.argv.slice(4);
8
9  function grep(searchString, flags, files) {
10     files.forEach(file => {
11         const fileContent = fs.readFileSync(file, 'utf8');
12         const lines = fileContent.split('\n');
13
14         lines.forEach((line, index) => {
15             let matches = false;
16
17             if (line.includes(searchString)) {
18                 matches = true;
19
20                 if (flags.includes('-x') && line.trim() !== searchString) {
21                     matches = false;
22                 }
23
24                 if (matches && flags.includes('-v')) {
25                     console.log(`${file}:${index + 1}:${line}`);
26                 } else if (matches) {
27                     if (flags.includes('-n')) {
28                         console.log(`${file}:${index + 1}:${line}`);
29                     } else {
30                         console.log(`${file}:${line}`);
31                     }
32                 }
33             }
34         });
35     });
36 }
37 grep(searchString, flags, files);
```

## PREGUNTA 2:

Implementación de clases en JavaScript para Pokemon y Charizard

Esta pregunta (recordando la infancia) fue una practica de herencia y POO en JavaScript.

Se nos pide diseñar 2 clases, una llamada "Pokemon" y otra llamada "Charizard".

Estas clases (como se nos indica) deben hacer lo siguiente:

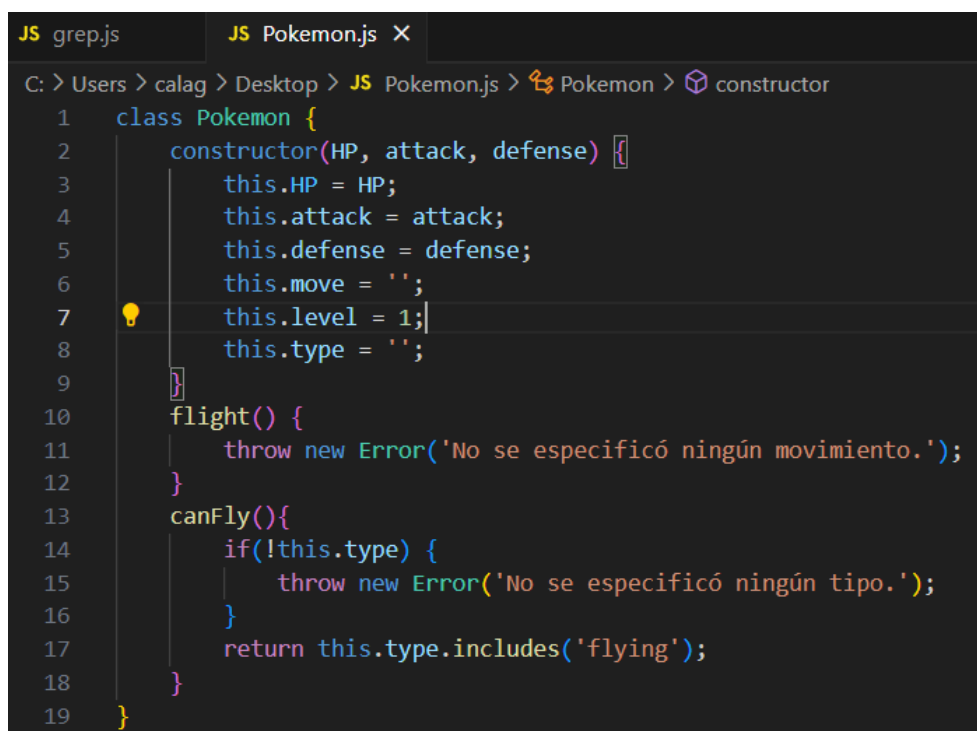
Clase Pokémon:

- El constructor toma 3 parámetros (HP, ataque, defensa)
- El constructor debe crear 6 campos (HP, ataque, defensa, movimiento, nivel, tipo). Los valores de (mover, nivelar, tipo) debe inicializarse en ("", 1, "").
- Implementa un método flight que arroje un error que indique que no se especifica ningún movimiento.
- Implementa un método canFly que verifica si se especifica un tipo. Si no, arroja un error. Si es así, verifica si el tipo incluye "flying". En caso afirmativo, devuelve verdadero; si no, devuelve falso

Clase Charizard:

- El constructor toma 4 parámetros (HP, ataque, defensa, movimiento).
- El constructor configura el movimiento y el tipo (para "disparar/volar") además de establecer HP, ataque y defensa como el constructor de superclase.
- Sobreescribe el método fight. Si se especifica un movimiento, imprime una declaración que indique que se está utilizando el movimiento y devuelve el campo de ataque. Si no arroja un error.

Entonces, al realizar la implementación en JavaScript siguiendo las indicaciones tenemos:



```
JS grep.js JS Pokemon.js X
C: > Users > calag > Desktop > JS Pokemon.js > Pokemon > constructor
1 class Pokemon {
2   constructor(HP, attack, defense) {
3     this.HP = HP;
4     this.attack = attack;
5     this.defense = defense;
6     this.move = '';
7     this.level = 1;
8     this.type = '';
9   }
10  flight() {
11    throw new Error('No se especificó ningún movimiento.');
```

```

21 class Charizard extends Pokemon {
22   constructor(HP, attack, defense, move) {
23     super(HP, attack, defense);
24     this.move = move;
25     this.type = 'flying';
26   }
27   fight() {
28     if (this.move) {
29       console.log('Se está utilizando el movimiento ${this.move}');
30       return this.attack;
31     } else {
32       throw new Error('No se especificó ningún movimiento.');

```

Ejemplo de uso:

```
const charizard = new Charizard(100, 80, 75, 'Fire Blast');
```

```
console.log(charizard.canFly()); // Devuelve true
```

```
console.log(charizard.fight()); // Imprime el movimiento y devuelve el ataque
```

### PREGUNTA 3:

Problema con el enfoque actual y aplicación de inyección de dependencia en Ruby.

Se nos brinda un enfoque simple para implementar la clase `MonthlySchedule` que conoce el horario de trabajo para un mes determinado, que inicializa con el año y el mes.

Se nos pregunta:

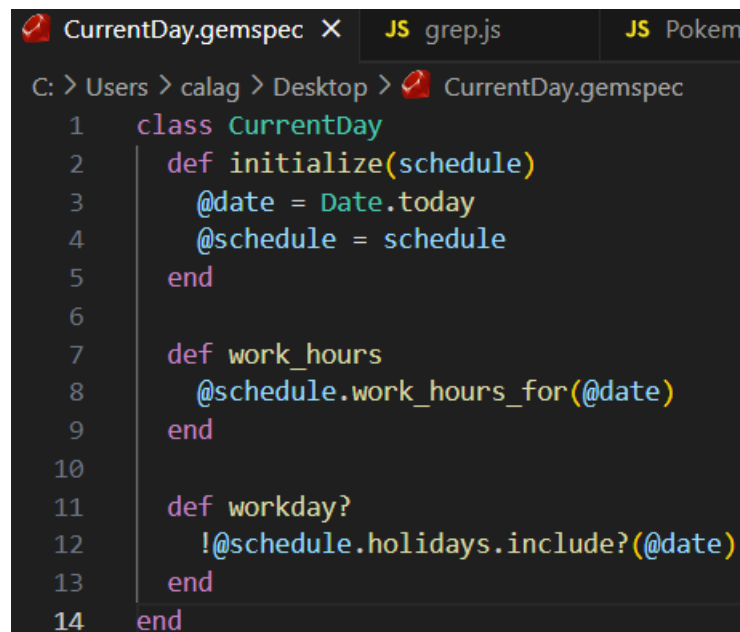
“¿Cuál es el problema con este enfoque dado, cuando quieres probar el método `workday?`

Utiliza la inyección de dependencia aplicado al siguiente código”

Respuesta:

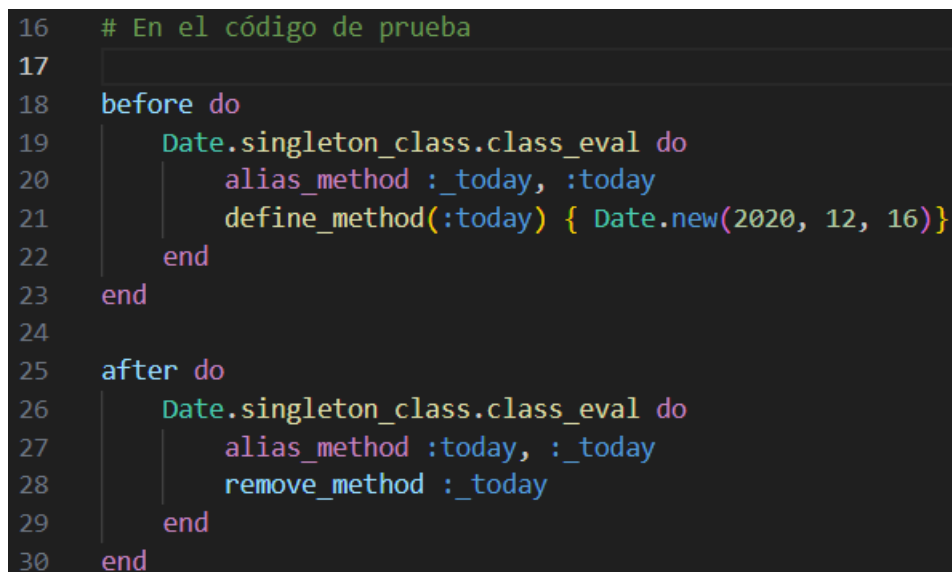
El problema con el enfoque actual en Ruby es que la clase `CurrentDay` instancia directamente `MonthlySchedule`, creando una dependencia rígida. Para probar el método `workday?`, se necesita modificar la fecha actual, lo cuál es incómodo.

Aplicando inyección de dependencias para facilitar las pruebas:



```
CurrentDay.gemspec X JS grep.js JS Pokemo
C: > Users > calag > Desktop > CurrentDay.gemspec
1 class CurrentDay
2   def initialize(schedule)
3     @date = Date.today
4     @schedule = schedule
5   end
6
7   def work_hours
8     @schedule.work_hours_for(@date)
9   end
10
11  def workday?
12    !@schedule.holidays.include?(@date)
13  end
14 end
```

En el código de prueba:



```
16 # En el código de prueba
17
18 before do
19   Date.singleton_class.class_eval do
20     alias_method :_today, :today
21     define_method(:today) { Date.new(2020, 12, 16) }
22   end
23 end
24
25 after do
26   Date.singleton_class.class_eval do
27     alias_method :_today, :today
28     remove_method :today
29   end
30 end
```

¿Qué sucede en JavaScript con el DIP en este ejemplo?

En JavaScript, el DIP se trata de cómo se estructuran las dependencias entre los módulos. Si tienes módulos de alto nivel que dependen de abstracciones y las implementaciones concretas dependen de esas abstracciones, estás siguiendo el principio de inversión de dependencia.

Podemos aplicar DIP utilizando patrones como inyección de dependencia manualmente pasando objetos o funciones a otros módulos, permitiendo una mayor flexibilidad en el intercambio de comportamiento y evitando dependencias rígidas.

## PREGUNTAS

1. Atributos privados en objetos User mediante clausuras. Crear un constructor de objetos 'User' que utilice clausuras para mantener la contraseña privada.

```
C: > Users > calag > Desktop > JS User.js > ...
1  function User(username, password) {
2      let _password = password; // Atributo privado
3
4      return {
5          username: username,
6          checkPassword: function(inputPassword) {
7              return _password === inputPassword;
8          }
9      };
10 }
11
12 const user = new User('JhonDoe', 'password123');
13 console.log(user.username); // Acceso público
14 console.log(user._password); // No se puede acceder (privado)
15 console.log(user.checkPassword('password123')); // Verificar contraseña
```

Este código crea un objeto 'user' que mantiene la contraseña como un atributo privado y expone un método 'checkPassword' para verificarla.

2. Validación de ActiveForm y generación de código JavaScript. Se extiende la función de validación para generar automáticamente código JavaScript que valide las entradas del formulario.
3. Identificar filas ocultas en una tabla utilizando JavaScript del lado cliente. ¿Cómo identificaría las filas que están ocultas utilizando sólo JavaScript del lado cliente?

```
1  const hiddenRows = Array.from(document.querySelectorAll('tr[style*="display:none"]'));
2  hiddenRows.forEach(row => {
3      // identificar filas ocultas y realizar operaciones
4  });
```

Este código busca filas ocultas en una tabla HTML mediante JavaScript del lado cliente.

- Implementar un conjunto de casillas de verificación para filtrar películas. Crear casillas de verificación para cada clasificación de película (G, PG, etc.) que al ser desmarcadas oculten las películas correspondientes.

```
1 <!-- HTML -->
2 <input type="checkbox" id="G" name="rating" value="G" checked>
3 <label for="G">G</label>
4 <input type="checkbox" id="PG" name="rating" value="PG" checked>
5 <label for="PG">PG</label>
```

```
1 <!-- JavaScript -->
2 const checkboxes = document.querySelectorAll('input[name="rating"]');
3 checkboxes.forEach(checkbox => {
4     checkbox.addEventListener('change', function() {
5         const rating = this.value;
6         const movies = document.querySelectorAll('tr[data-rating="${rating}"]');
7         movies.forEach(movie => {
8             movie.style.display = this.checked ? 'table-row' : 'none';
9         });
10    });
11 });
```

Esto crea un conjunto de casillas de verificación que filtran películas por su clasificación

- Ajax para menús de cascada basados en asociaciones has\_many. Usar AJAX para generar menús desplegables donde seleccionar una opción de un menú (A) actualice las opciones del otro menú (B) correspondientes.
- Implementar una memoria caché en JavaScript para solicitudes AJAX repetidas. Desarrollar una lógica en JavaScript para almacenar en caché la información de la película obtenida en una solicitud AJAX, evitando solicitudes adicionales si la información ya está en caché.

```
1 const cache = {};
2
3 function fetchData(movieId) {
4     if (cache[movieId]) {
5         return Promise.resolve(cache[movieId]);
6     } else {
7         return fetch(`/movies/${movieId}`)
8             .then(response => response.json())
9             .then(data => {
10                 cache[movieId] = data;
11                 return data;
12             });
13     }
14 }
```

Esta lógica utiliza un objeto 'cache' para almacenar la información de las películas obtenidas por AJAX, evitando solicitudes repetidas.