[OSAKamp] ASOCIACIÓN DE EMPRESAS DE TECNOLOGÍAS LIBRES Y CONOCIMIENTO ABIERTO DE EUSKADI

03 - POO

Teoria

CODE4JOBS

Bizkaia
toru aldundia
diputación foral

[esle]

Introducción

La Programación Orientada a Objetos (POO) es un paradigma de programación basado en los conceptos de Objetos. Eso significa que todo lo que logramos en el lenguaje OOP es a través de objetos. El objetivo principal del desarrollo de la programación orientada a objetos era organizar la estructura del código. Con la programación orientada a objetos se puede escribir código más modular y fácil de mantener. Puedes asociar el código con entidades del mundo real.

Al utilizar la programación orientada a objetos, te aseguras de que sólo los miembros permitidos de un código sean accesibles a otros. Esto hace que tu código sea totalmente seguro frente a accesos no autentificados (dentro del código). Ahora veamos los conceptos principales de la Programación Orientada a Objetos paso a paso.

Objeto

Como ya he mencionado anteriormente, los Objetos son como entidades de la vida real. Tienen sus propiedades y métodos.

Considera un coche como un objeto. El coche tiene muchas características como el color, el nombre de la empresa, el nombre modal y el precio, etc. En un coche, podemos realizar acciones como arrancar, frenar y parar. Aquí las características de un coche son propiedades, y las acciones son métodos.

Si estás usando javascript por un tiempo, puede que uses objetos muchas veces en tu código, pero tal vez no de una manera OOP.

Déjame crear un objeto usuario aquí.

```
const user = {
    name: 'Peio Murguia',
    userName: 'Kazpiter',
    password: 'contraseña:)',
    login: function(userName, password) {
      if (userName === this.userName && password === this.password) {
        console.log('Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    },
  };


  user.login('Peio', 'Peio'); // Authentication Failed!!
  user.login('Kazpiter', 'contraseña:)'); // Login Successfully
```

El código anterior se explica por sí mismo. Estoy creando un objeto usuario con algunas propiedades y acciones que puede realizar. Vamos a entender algunos conceptos más de programación orientada a objetos.

Clase

Clase es un plano de una entidad de la vida real. Describe cómo será el objeto, qué características tiene y qué tipo de acciones podemos realizar sobre él. La clase es sólo una plantilla. No se puede realizar ninguna acción sobre ella. Instanciamos el objeto a partir de una clase. Podemos crear muchas instancias de una clase.

Veamos un ejemplo.

```
class User {
    #password;
    constructor(name, userName, password) {
      this.name = name;
      this.userName = userName;
      this.#password = password;
    }

    login(userName, password) {
      if (userName === this.userName && password === this.#password) {
        console.log('Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    }

    setPassword(newPassword) {
      this.#password = newPassword;
    }
  };

  const Peio = new User('Peio Murguia', 'Kazpiter', 'contraseña:)');
  const js = new User('JavaScript', 'js', 'python:)');


  Peio.login('Kazpiter', 'contraseña:)'); // Login Successfully
  js.login('js', 'python:)'); // Login Successfully

  console.log(Peio.name); // Peio Murguia
  console.log(Peio.password); // undefined
  console.log(Peio.#password); // Syntax Error

  Peio.setPassword('new_contraseña:)');
  Peio.login('Kazpiter', 'contraseña:)'); // Authentication Failed!!
  Peio.login('Kazpiter', 'new_contraseña:)'); // Login Successfully
```

Aquí he creado una clase llamada Usuario, que tiene algunas propiedades y métodos. Luego estoy creando instancias de la clase usando new User() y pasando los valores de las propiedades requeridas.

¿Has visto un método constructor que nunca hemos llamado en nuestro código?

Cuando creamos un objeto a partir de una clase usando la palabra clave new, javascript internamente llama al método constructor que inicializa las propiedades públicas y privadas de una clase. El objeto aquí puede acceder a todas las propiedades públicas y métodos de una clase.

Por defecto, todas las propiedades declaradas en la clase son públicas, lo que significa que puedes llamarlas y modificarlas desde fuera de la clase. Puedes declarar propiedades públicas dentro o fuera del constructor. Aquí name y userName son propiedades públicas, pero password es una propiedad privada. Hash(#) indica que esta propiedad es privada para la clase y sólo los métodos que se declaran dentro de la clase pueden acceder a ella. Las propiedades privadas deben ser declaradas antes de ser utilizadas.

Cuando intenté imprimir la contraseña, obtuve undefined ya que no tengo ningún miembro llamado 'contraseña', entonces lo intenté con '#contraseña' que me dio un error de sintaxis porque '#contraseña' es privada.

Para imprimir/modificar las propiedades privadas, necesitamos métodos getter/setter. Aquí he creado un método que establece la nueva contraseña. Los siguientes conceptos son los cuatro pilares del lenguaje OOP.

Encapsulación

La encapsulación se define como la unión de datos y métodos en una sola unidad para protegerlos del acceso exterior. Al igual que una pastilla contiene la medicación dentro de su cubierta.

En el contexto de una clase, algunas propiedades no son accesibles directamente desde fuera de la clase. Es necesario llamar al método responsable de las propiedades.

En el ejemplo anterior, ya hemos utilizado la encapsulación. Enlazamos (lógicamente) la propiedad privada password con un método público setPassword(). También tenemos un método getter, que devuelve el valor actual de una propiedad privada.

Abstracción

La gente a menudo confunde encapsulación con abstracción. La abstracción es un paso adelante de la encapsulación. La abstracción se define como mostrar sólo las cosas esenciales y ocultar la implementación interna.

Tomemos el ejemplo de un coche. En un coche, podemos realizar algunas acciones como arrancar, frenar y parar. Cada vez que se llama a una de estas acciones, se obtiene un resultado. Estas acciones tienen ciertas sub-acciones que están ocultas para ti, pero no necesitas preocuparte por esas sub-acciones.

Así es como una empresa de automóviles utiliza una abstracción de la funcionalidad para dar a sus clientes una experiencia sin problemas.

Tomemos otro ejemplo de abstracción. Supongamos que usted está utilizando algún componente react de terceros para su proyecto front-end. Este componente proporciona muchos props y métodos para su personalización. Este componente no es mágico, internamente utiliza las mismas etiquetas HTML, CSS y javascript. Pero ahora no necesitas

preocuparte por esas cosas. Solo necesitas establecer props y llamar métodos basados en tus requerimientos. Eso es una abstracción.

Vamos a codificar 😍

```
class User {
    name;
    email;
    #password;
    constructor() {}

    #validateEmail(email) {
      // check email is valid or not.
      return true;
    }

    #validatePassword(password) {
      // check password is satisfying the minimum requirements or not.
      return true;
    }

    signUp(name, email, password) {
      let isValidated = false;
      isValidated = this.#validateEmail(email);
      isValidated &&= this.#validatePassword(password);

      if (isValidated) {
        this.name = name;
        this.email = email;
        this.#password = password;
        // add user in your db.
        console.log('User registered successfuly');
      } else {
        console.log('Please enter correct Details!!');
      }
    }

    login(email, password) {
      if (email === this.email && password === this.#password) {
        console.log('Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    }

    #isRegisteredUser(email) {
      // check user is registered or not.
      return true;
    }
```

```
    resetPassword(email, newPassword) {
      if (this.#isRegisteredUser(email)) {
          this.#password = newPassword;
          console.log('Operation performed successfully');
      }
      else {
        console.log('No account found!');
      }
    }
  };

  const Peio = new User();
  Peio.signUp('Peio Murguia', 'nm@gmail.com', 'contraseña:)'); // User
registered successfuly

  Peio.#validateEmail('nm@gmail.com'); // Syntax Error.

  Peio.login('nm@gmail.com', 'contraseña:)'); // Login Successfully
  Peio.resetPassword('nm@gmail.com', ''); // Operation performed
successfully
```

En el ejemplo anterior, hemos introducido algunos métodos privados. Los métodos están haciendo algún trabajo y no están expuestos al exterior de la clase.

Estos métodos son llamados por los métodos disponibles públicamente.

Como desarrollador, sólo tengo que dar los detalles que he recibido de la interfaz de usuario y llamar al método responsable.

En lenguajes de programación orientada a objetos como Java, tenemos el concepto de clases abstractas e interfaces. Eso no es posible en javascript.

De lo contrario, podemos crear una clase abstracta y esa clase puede ser utilizada por otra clase para lograr una funcionalidad similar.

Así que básicamente podemos decir que estamos utilizando la encapsulación para lograr la abstracción 😊.

Herencia

Cuando una clase deriva las propiedades y métodos de otra clase se llama herencia en OOP. La clase que hereda la propiedad se conoce como subclase o clase hija y la clase cuyas propiedades se heredan se conoce como superclase o clase padre.

¿Por qué necesitamos la herencia?

La herencia es un concepto muy importante en la programación orientada a objetos. La principal ventaja de la herencia es la reutilización. Cuando una clase hija hereda de una clase padre no necesitamos escribir el mismo código de nuevo. Resulta muy fiable cuando necesitamos hacer algún cambio en las propiedades, basta con cambiarlo en una clase padre y

todas las clases hijas heredarán automáticamente el cambio. La herencia también promueve la legibilidad del código.

Vamos a codificar...

```
class User {
    #password;
    constructor(email, password) {
      this.email = email;
      this.#password = password;
    }

    login(email, password) {
      if (email === this.email && password === this.#password) {
        console.log('Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    }

    resetPassword(newPassword) {
      this.#password = newPassword;
    }

    logout() {
      console.log('Logout Successfully');
    }
  }

  class Author extends User {
    #numOfPost;

    constructor(email, password) {
      super(email, password);
      this.#numOfPost = 0;
    }

    createPost(content) {
      // add content to your DB. :)
      this.#numOfPost++;
    }

    getNumOfPost() {
      return this.#numOfPost;
    }
  }

  class Admin extends User {
    constructor(email, password) {
```

```
      super(email, password);
    }

    removeUser(userId) {
      // remove this userId from your DB.
      console.log('User Removed successfully.');
    }
  }

  const Peio = new Author('nm@gmail.com', 'contraseña:)');
  Peio.login('nm@gmail.com', 'contraseña:)');
  Peio.createPost('Escribamos que las rosas son rojas y el cielo es azul
:)');
  Peio.createPost('aquí no sé qué más poner');
  console.log(Peio.getNumOfPost()); // 2

  const json = new Admin('jason@gmail.com', '[Object] [object]');
  json.login('jason@gmail.com', '[Object] [object]');
  json.resetPassword('{id: 1}');
  json.login('jason@gmail.com', '{id: 1}');
  json.removeUser(12);
```

En el ejemplo anterior, las clases Author y Admin heredan la propiedad de la clase User usando las palabras clave extends y super.

La palabra clave extends se utiliza para establecer una relación padre-hijo entre dos clases. En el primer caso, el Autor se convierte en subclase y el Usuario en clase padre.

La subclase tiene acceso a todos los miembros públicos y protegidos de la superclase. Además, puede tener sus propias propiedades y métodos. Así es como podemos lograr la reutilización a través de la herencia.

La palabra clave super es una palabra clave especial. Llamar a super en el constructor del hijo invoca al constructor padre. Así es como estamos inicializando las propiedades en las clases Author y Admin.

La clase hija también puede sobrescribir los métodos de una clase padre. Esto introduce el concepto de polimorfismo.

Polimorfismo

Polimorfismo significa 'más de una forma'. Como nosotros, Los ingenieros de software podemos trabajar en el frontend, backend, DevOps e incluso pruebas. 😅

El polimorfismo tiene dos tipos.

1. Polimorfismo en tiempo de compilación
2. Polimorfismo en tiempo de ejecución

La sobrecarga de funciones es un tipo de polimorfismo en tiempo de compilación. Aquí, estamos creando más de una función con el mismo nombre y diferentes parámetros o tipos.

La sobrecarga de funciones no está soportada en JavaScript porque si creas funciones con el mismo nombre, Javascript sobreescribirá la última función definida con funciones anteriores.

La sobrecarga de métodos es un tipo de polimorfismo en tiempo de ejecución. ¿Recuerdas que te dije que puedes anular los métodos de la clase padre en la clase hija? Eso es sobreescritura de métodos.

Tomemos un ejemplo.

```
class User {
    constructor(email, password) {
      this.email = email;
      this.password = password;
    }

    login(email, password) {
      if (email === this.email && password === this.password) {
        console.log('Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    }
  }

  class Author extends User {
    #numOfPost;

    constructor(email, password) {
      super(email, password);
      this.#numOfPost = 0;
    }

    createPost(content) {
      // add content to your DB. :)
      this.#numOfPost++;
    }

    getNumOfPost() {
      return this.#numOfPost;
    }
  }

  class Admin extends User {
    constructor(email, password) {
      super(email, password);
    }

    login(email, password) {
      // add extra layer of security as this is an admin account.
```

```
      const isValidAdmin = true; // we can have some 2FA type security
check here.
      if (email === this.email && password === this.password &&
isValidAdmin) {
        console.log('Admin Login Successfully');
      } else {
        console.log('Authentication Failed!!');
      }
    }

    removeUser(userId) {
      // remove this userId from your DB.
      console.log('User Removed successfully.');
    }
  }

  const Peio = new Author('nm@gmail.com', 'contraseña:)');
  Peio.login('nm@gmail.com', 'contraseña:)'); // Login Successfully

  const json = new Admin('jason@gmail.com', '[Object] [object]');
  json.login('jason@gmail.com', '[Object] [object]'); // Admin Login
Successfully
```

Aquí, tanto el Autor como el Administrador heredan de la clase Usuario. Ambas clases tienen el método login de la clase User. Ahora necesito algún nivel extra de verificación para la cuenta admin, así que he creado un método de login en la clase Admin. Este sobrescribirá el método de inicio de sesión de la clase padre.

Cuando un objeto de la clase Admin llame al método login, invocará una llamada de función al método login de la clase Admin.

Así es como hemos conseguido el polimorfismo utilizando la sobreescritura de métodos.

Y ya está. Hemos cubierto todos los conceptos de POO con JavaScript. 😍

Nota: Toda la información anterior se basa en mi knowlege y la investigación. Si encuentras algo mal aquí, por favor corrígeme en la sección de comentarios. Feliz aprendizaje 🙂