Aplicando lo aprendido 0/3

Alumno: Eber Chiecher
Universidad:Unvime
Asignatura: Paradigmas de Programación

Aplicando lo aprendido 0)

1. Generalización simbólica: ¿Cuáles son las reglas escritas del lenguaje?
Lenguaje java
Respuesta: Las reglas escritas del lenguaje: es fuertemente tipado, eso significa que se debe definir el tipo de la variable que se crea de la misma forma se debe definir el tipo de los métodos. No se puede ingresar un tipo de dato x en una variable de tipo y, en la variable de determinado tipo se debe ingresar solo ese tipo. En java cuando igualas un objeto lo que estás haciendo es que apunte a la misma dirección de memoria que el objeto que estas igualando, Ejemplo: nuevo= Objeto_1; nuevo apunta a la misma dirección de Objeto_1 si modifico Objeto_1 se va a modificar nuevo y viceversa.

2. Creencias de los profesionales: ¿Qué características particulares del lenguaje se cree que sean "mejores" que en otros lenguajes?

Respuesta:Java al ser un lenguaje tan estricto, nos ahorra muchos problemas de compilación y seguridad. Otra cosa que lo beneficia de otros lenguajes es su máquina virtual(JVM) esta permite que sea multiplataforma permitiendo que un código se ejecute en diferentes entornos.

3. Los valores de un grupo acerca de lo que es más importante.
Respuesta:Java al tener una larga historia, sus bibliotecas estándares son muy robustas, ahorrando a los desarrolladores el tener que crear ciertas implementaciones. La modularización de java, el hecho que sea orientado a objetos, permite la reutilización del código y que sea modularizable y escalable.
4. Ejemplares: ¿Qué clase de problemas pueden resolverse más fácilmente en el lenguaje?
Respuesta: Java es bueno para resolver problemas, que requieren especificar paso a paso que se debe hacer, esto va acorde a su paradigma imperativo. Estos problemas pueden ser algoritmos de ordenamiento, búsqueda y manipulación de estructuras de datos.

Aplicando lo aprendido 1)

1. Generalizaciones simbólicas:

    JavaScript, en el contexto de la programación estructurada, sigue reglas y leyes específicas. Se basa en la sintaxis estructurada para la creación de programas. Por ejemplo, la definición de funciones, estructuras de control de flujo (if, else, switch, while, for), y la manipulación de variables siguen reglas bien definidas.

2. Creencias de la comunidad de profesionales:

   La comunidad de desarrolladores de JavaScript en el paradigma de programación estructurada tiende a seguir prácticas como la modularización del código mediante funciones, el uso de estructuras de control de flujo para la toma de decisiones y la repetición, y la importancia de mantener el código claro y legible. Además, las prácticas de depuración y pruebas unitarias son valoradas para garantizar la calidad del código.

3. Valores del grupo acerca de lo que es más importante:

   En la programación estructurada en JavaScript, se valora la claridad y la legibilidad del código. La modularidad y la reutilización del código a través de funciones son aspectos importantes. Además, la eficiencia y el rendimiento también pueden ser valores importantes dependiendo del contexto de la aplicación.

4. Ejemplares, incluyendo los problemas a ser resueltos con sus soluciones:

   Los problemas resueltos con JavaScript en el paradigma de programación estructurada incluyen tareas como la manipulación de datos, la interacción con el DOM (Document Object Model) en el contexto del desarrollo web, la implementación de algoritmos y la gestión de eventos. Ejemplos específicos podrían ser la implementación de un algoritmo de ordenamiento, la manipulación de formularios en una página web, o la validación de datos de entrada.

Aplicando lo aprendido 2)

TypeScript desde la perspectiva de los cuatro componentes de un paradigma según Kuhn:

1) Generalización simbólica: ¿Cuáles son las reglas escritas del lenguaje?

   Reglas Escritas del Paradigma:

   TypeScript es una extensión tipada de JavaScript, lo que significa que comparte muchas de las reglas y características de JavaScript, pero añade un sistema de tipos estáticos. Las reglas escritas del paradigma en TypeScript incluyen la definición de tipos de datos, interfaces, clases, y el uso de anotaciones de tipos para proporcionar información estática sobre el código.

2) Creencias de los profesionales: ¿Qué características particulares del lenguaje se cree que sean "mejores" que en otros lenguajes?

   Typescript cuenta con una comunidad grande en desarrollo, cuenta con bibliotecas y herramientas en constante crecimiento.

   La adición de anotaciones de tipos y otras características del sistema de tipos puede mejorar la legibilidad del código. Los profesionales valoran la claridad que aporta TypeScript al código, especialmente en proyectos grandes y complejos.

   Una de las características más destacadas de TypeScript es su sistema de tipos estáticos. Los profesionales suelen valorar la capacidad de detectar errores en tiempo de compilación, lo que proporciona una capa adicional de seguridad y ayuda a evitar problemas comunes durante la ejecución del programa.

3) Valores del Grupo:

   Importancia del Mantenimiento del Código: Dada la naturaleza tipada de TypeScript, hay una valoración significativa en la capacidad de mantener y escalar proyectos de manera eficiente. La prevención de errores en tiempo de compilación se considera fundamental para la calidad del código.

4) Ejemplares (Problemas y Soluciones):

   Problemas a Ser Resueltos:

   Manejo de Grandes Códigos: TypeScript es especialmente útil en proyectos grandes donde el sistema de tipos puede ayudar a prevenir errores comunes y mejorar la legibilidad del código.

Aplicando lo aprendido 3)

Ejercicio 1

JavaScript desde la perspectiva de los cuatro componentes de un paradigma POO basada en prototipos según Kuhn:

1. Generalización Simbólica:

   Reglas Escritas del Paradigma: En JavaScript, el paradigma predominante es la programación orientada a objetos basada en prototipos. Las reglas incluyen la creación de objetos mediante

funciones constructoras, el uso de prototipos para herencia, y la manipulación de objetos y propiedades.

2. Creencias de los Profesionales:

    Características Valoradas:

        Flexibilidad: JavaScript es conocido por su flexibilidad y dinamismo. Los profesionales valoran la capacidad de adaptarse fácilmente a diferentes paradigmas de programación, incluida la programación funcional y la orientada a eventos.

        Prototipos como Mecanismo de Herencia: Aunque diferente de la herencia clásica basada en clases, el uso de prototipos es valorado por algunos desarrolladores por su simplicidad y flexibilidad.

3. Valores del Grupo:

    Importancia de la Interactividad en el Navegador: JavaScript se ha vuelto fundamental para la interactividad en los navegadores web. Los profesionales valoran su capacidad para mejorar la experiencia del usuario en el lado del cliente.

4. Ejemplares (Problemas y Soluciones):

    Problemas a Ser Resueltos:

        Desarrollo Web: JavaScript es esencial en el desarrollo web para manipular el DOM, responder a eventos del usuario y realizar solicitudes asíncronas al servidor.

        Desarrollo Front-End: En el desarrollo front-end, JavaScript se utiliza para construir interfaces de usuario interactivas y dinámicas.

En resumen, JavaScript, desde el paradigma de programación orientada a objetos basada en prototipos, destaca por su flexibilidad y su importancia en el desarrollo web. Los profesionales valoran su capacidad para adaptarse a diferentes estilos de programación y su papel crítico en la creación de experiencias interactivas en el navegador. La orientación a objetos en JavaScript se realiza mediante prototipos, lo que, aunque diferente de la herencia basada en clases, ha demostrado ser eficaz en muchos escenarios de desarrollo web.

Ejercicio 3

(proyecto to list con prototipos)

abstracción

Al crear el objeto prototypo tarea, estoy usando la propiedad de abstracción al definir los atributos del objeto, porque estoy abstrayendo propiedades de una tarea de la vida real.
tarea.js
linea 5

```
///Objeto prototipo
var Tarea ={
        titulo:"",
        descripcion:"",
        estado: 1,
        creacion:new Date(),
        ultima_edicion: new Date(),
        vencimiento:new Date(),
        dificultad:0
}
```

polimorfismo

En este caso no use polimorfismo, el polimorfismo está fuertemente relacionado con la herencia, en este caso tengo una sola jerarquia que es tarea(sin contar Object). Para usar polimorfismo deberia agregar otra jerarquia, ejemplo, tarea_pendiente que sea una copia de tarea, mas sus propias propiedades(herencia). Con ella podria aplicar polimorfismo, haciando que al llamar un metodo de una instancia tarea, utilizando una instancia tarea_pendiente se comporte de forma distinta. En términos más sencillos, significa que un objeto puede ser tratado como si fuera de un tipo diferente al que realmente tiene.

Herencia

Tarea.js

linea 16

```
function crearTarea(titulo, descripcion, vencimiento, dificultad) {
```

var nuevo=Object.create(Tarea);///nuevo va a ser una copia de Tarea

///asigno los valores ingresados atraves de parametros al objeto

nuevo.titulo=titulo;

nuevo.descripcion=descripcion;

nuevo.vencimiento=vencimiento;

nuevo.dificultad=dificultad;

nuevo.creacion=new Date();

nuevo.ultima_edicion=new Date();

return nuevo;///Devuelvo objeto nuevo

}

En este caso estoy usando herencia ya que en esta funcion estoy creando copias(instancias) del objeto tarea, estas copias heredan todas las propiedades de tarea. Ademas tarea hereda de Object quien es su prototipo

cadena prototipica: Object→tarea→instancia_de_tarea

Encapsulamiento

En este caso no aplique encapsulamiento, no hay ningun tipo de restriccion en las propiedades de los Objetos, se puede acceder y modificar las propiedades de los objetos desde cualquier entorno.

Cohecion

Se refiere a que tan bien definida esta la funcion o el objeto. Ejemplo:

Tarea.js

linea 45

```
function editar(x) {

x.titulo = readline.question("Titulo: \n");

x.descripcion = readline.question("Descripcion: \n");

x.estado=5;

while (x.estado != 1 && x.estado != 2 && x.estado != 3) {

///utilizo un console.log en vez de poner la descripcion en el questionInt, porque si no no muestra los emojis

console.log(`${kleur.red(`1. Pendiente ${emoji.get('tada')}`)}\n${ kleur.yellow(`2. En curso ${emoji.get('hourglass')}`)}\n${kleur.green(`3. Finalizado ${emoji.get('white_check_mark')}`)}`);

x.estado = readline.questionInt();

}

x.ultima_edicion = new Date();

do {

const fechaTexto = readline.question('Ingresa fecha limite (YYYY-MM-DD): ');

x.vencimiento = new Date(fechaTexto);

} while (isNaN(x.vencimiento.getTime()));

///utilizo un console.log en vez de poner la descripcion en el questionInt, porque si no no muestra los emojis

x.dificultad=5;

while (x.dificultad != 1 && x.dificultad != 2 && x.dificultad != 3) {

///utilizo un console.log en vez de poner la descripcion en el questionInt, porque si no no muestra los emojis

console.log(`${kleur.green(`1.Facil ${emoji.get('smile')}`)}\n${kleur.yellow(`2.Medio ${emoji.get('neutral_face')}`)}\n${kleur.red(`3.Dificil ${emoji.get('rage')}`)}`);
```

```
x.dificultad = readline.questionInt();

}

console.log("Datos guardados!\n");

}
```

esta funcion recive como parametro un objeto tipo tarea, luego va editando cada uno de sus atributos y tambien se encarga de validarlos. En este caso para que haya una mayor cohecion se podria crear otra funcion que tenga el rol especifico de validar datos ingresados por teclados para que asi, la funcion editar solo se encargue de una sola funcion en especifico. Ya que para que la funciones tengan mayor cohecion se necesita que la función realice una tarea específica y bien definida y que todos los elementos dentro de la función están directamente relacionados con la tarea principal.

Una mala cohecion en un objeto se refiere al sentido de sus propiedades ejemplo: Si a mi objeto tarea le huebiera agregado como atributos, cantidad de ruedas, velocidad_maxima y color; Hubiera bajado el nivel de cohecion ya que no tendria sentido esos atributos para mi objeto tarea.

Ejercicio 2 (proyecto calculadora con clases)

abstracción

```
class Calculadora{



Constructor(){

this.a=0;

this.b=0;

}
```

```
///metodos



suma(a,b){

this.a=a;

this.b=b;

return a+b;



}

resta(a,b){

this.a=a;

this.b=b;

return a-b;

}

multiplicacion(a,b){

this.a=a;

this.b=b;

return a*b;

}

division(a,b){

this.a=a;

this.b=b;

return a/b;
```

}

}

Se aplica la abstraccion ya que se abstrae las propiedades de una calculadora de la vida real: una calculadora suma, resta, multiplica y divide. Ademas de tener dos valores de ingreso.

Encapsulamiento

En este caso no aplique encapsulamiento, no hay ningun tipo de restriccion en las propiedades de los Objetos, se puede acceder y modificar las propiedades de los objetos desde cualquier entorno.

Cohecion

Aqui hay un alto nivel de cohecion ya que cada metodo tiene una sola tarea en especifico, que se cumple gratificantemente, todos sus elementos estan fuertemente relacionados en la tarea a realizar. El objeto calculadora tambien tiene una alta cohecion ya que todos sus propiedades estan fuertemente relacionadas.

Polimorfismo

No se aplica polimorfismo por lo mencionado en el proyecto anterior

Herencia

Se aplica herencia al crear una instancia de la clase calculadora:

index.js

linea 16

x=new calculadora();