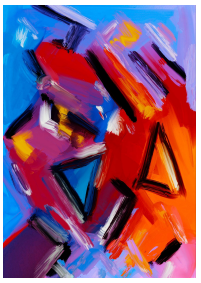
Clase III

OTROS RECURSOS, EXCEPCIONES Y ARCHIVOS



IEEE ITBA
STUDENT BRANCH

Índice

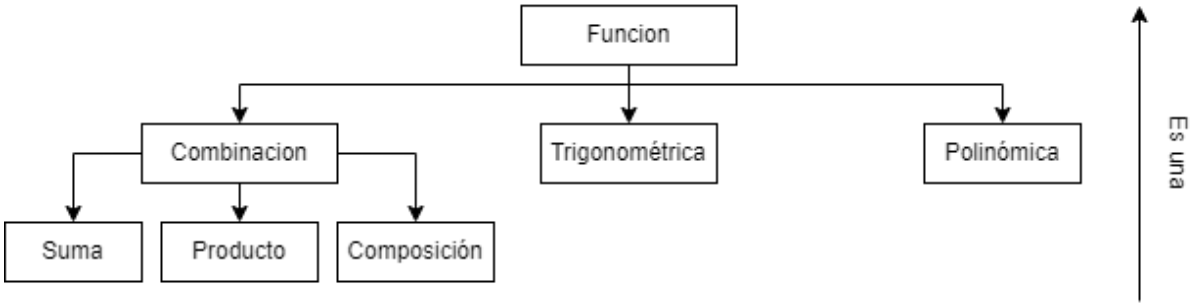
Extensión de Clases

Introducción

Continuando con la clase pasada, donde llegamos a ver clases y una implementación bastante simple, hoy veremos extensión de clases e interfaces. Se podrían preguntar ¿qué significa esto?

Imaginemos que queremos tener una representación de funciones matemáticas. Algunas funciones matemáticas son la suma, el producto, polinómicas, etc. También tenemos funciones compuestas, que son, precisamente, una composición de funciones. Un claro ejemplo sería la función *sin(2x)*, donde están compuestas las funciones *sin(x)* y *2x*.

Lo que tienen todas las funciones en común, es que se pueden evaluar en un punto (x). Debido a esto, podemos decir que tienen una funcionalidad en común, el método evaluar(x). Por lo tanto, habría que pensar en que habría que crear una interfaz en común para todas. De esta forma, nos quedará el siguiente cuadro de relaciones simples:



La relación *es una* es muy importante, ya que eso implica que *toma propiedades y atributos de*.

Para el modelado de este tipo de clases, existe una palabra reservada *extends*, que podrá hacer lo que buscamos. Recordar que la palabra reservada *implements* se utiliza para implementar la interfaz en nuestra clase.

Modelado de interfaz

Primero, vamos a modelar la interfaz IFuncion, la cual es bastante simple, ya que tiene únicamente un método. De nuevo, hacemos esto en un archivo nuevo, dentro de la carpeta de interfaces:

```
export default interface IFuncion {
      evaluar: (x: number) => number;
}
```

Clase Trigonométrica y Clase Polinómica

Luego, podríamos realizar un Enum para las tres operaciones trigonométricas básicas (seno, coseno, tangente), e implementar la Clase función trigonométrica[1]. Para la misma necesitamos un constructor, donde le pasaremos qué función trigonométrica representa, e implementar el método "evaluar":

```
import IFuncion from "../Interfaces/IFuncion";

export enum FuncionesTrigonometricas {
    Sin, Cos, Tan
}

export default class Trigonometrica implements IFuncion {
    private funcionTrigonometrica: FuncionesTrigonometricas ;

    constructor( funcionTrigonometrica: FuncionesTrigonometricas ) {
        this.funcionTrigonometrica = funcionTrigonometrica;
    }

    evaluar(x: number): number {
        switch(this.funcionTrigonometrica) {
            case FuncionesTrigonometricas.Sin: return Math.sin(x);
            case FuncionesTrigonometricas.Cos: return Math.cos(x);
            case FuncionesTrigonometricas.Tan: return Math.tan(x);
        }
    }
}
```

[1] La librería Math de TypeScript, ya incorporada directamente al lenguaje, se utiliza para realizar operaciones matemáticas, de las cuales utilizaremos solo sin, cos, tan y pow (potencia), pero tiene muchas funcionalidades más.

Para la Clase de funciones polinómicas, podríamos utilizar un arreglo de constantes, donde la posición en el arreglo indica el grado del término de la constante (la potencia[2]):

```
import IFuncion from "../Interfaces/IFuncion";

export default class Polinomica implements IFuncion {
    private pol: number[]; // pol[n-1]*x^n + ... + pol[1]*x + pol[0]

    constructor( pol: number []) {
        this.pol = pol;
    }

    evaluar(x: number): number {
        let res = 0;
        for( let i = 0; i < this.pol.length; i++ ) {
            res += Math.pow(x, i) * this.pol[i];
        }
        return res;
    }

}
```

Clase abstracta Combinación

Lo que tienen en común todas las combinaciones de funciones es que reciben dos funciones. A una la llamaremos función izquierda y, a la otra, función derecha. En vez de una interfaz Combinación, esta será una clase, debido a que se puede implementar un constructor en común.

La clase Combinación es la siguiente:

```
import IFuncion from "../Interfaces/IFuncion";

export default abstract class Combinacion implements IFuncion {
    protected f: IFuncion;
    protected g: IFuncion;

    constructor(f: IFuncion, g: IFuncion) {
        this.f = f;
        this.g = g;
    }
```

[2] *pow* viene de potencia, y el llamado a la función toma como primer argumento el número a potenciar y, como segundo argumento, la potencia a la que hay que elevar a ese número.

```
    abstract evaluar(x: number): number;
}
```

Van a poder nota que en el código usamos la palabra reservada *protected*.

La clase abstracta es un tipo especial de clases que no puede ser instanciada directamente, y donde los métodos pueden ser declarados como abstractos, para que sí o sí tenga que implementarlos la clase que herede sus atributos.

En TypeScript, una variable *protected* es una propiedad de una clase que se puede acceder dentro de la clase que lo define, así como en cualquier subclase que herede de esa clase. Sin embargo, las variables protegidas no se pueden acceder fuera de la clase o sus subclases.

Clases Suma, Producto y Composición

Finalmente, nos queda implementar las clases Suma, Producto, y Composicion, que heredan de Combinacion. Algo peculiar de esto es que, para implementarlas, no hay que recurrir a métodos sofisticados, sino que hay que delegar esto a las funciones izquierda y derecha:

```
import Combinacion from "./Combinacion";

export default class Suma extends Combinacion {
    evaluar(x: number): number {
        return this.f.evaluar(x) + this.g.evaluar(x); // f(x) + g(x)
    }

}
```

```
import Combinacion from "./Combinacion";

export default class Producto extends Combinacion {
    evaluar(x: number): number {
        return this.f.evaluar(x)*this.g.evaluar(x); // f(x)*g(x)
    }
}
```

```
import Combinacion from "./Combinacion";
```

```
export default class Composicion extends Combinacion {
    evaluar(x: number): number {
        return this.f.evaluar(this.g.evaluar(x)); // f(g(x))
    }
}
```

Testeo

Para poder testear que todo funcione correctamente, podríamos hacer un simple programa, donde probaremos con varios valores cuyo resultado al evaluar funciones específicas ya es conocido:

```
import Polinomica from "./Clases/Polinomica";
import Trigonometrica, {FuncionesTrigonometricas} from
"./Clases/Trigonometrica";
import Composicion from "./Clases/Composicion";
import Suma from "./Clases/Suma";
import Producto from "./Clases/Producto";

function main() {
    // Definiendo funciones de formato aleatorio
    // f(x) = 3
    let constante3 = new Polinomica([3]);
    // f(x) = x^2
    let cuadrado= new Polinomica([0,0,1]);
    // f(x) = cos(x)
    let coseno = new Trigonometrica(FuncionesTrigonometricas.Cos);
    // f(x) = (cos(x))^2
    let cosenoCuadrado = new Producto(coseno, coseno);
    // f(x) = sin(x)
    let seno = new Trigonometrica(FuncionesTrigonometricas.Sin);
    // f(x) = (sin(x))^2
    let senoCuadrado = new Composicion(cuadrado, seno);
    // f(x) = 3sin(x)
    let constantePorSeno = new Producto(seno, constante3);
    // f(x) = 2x^3 + sin(x)
    let polinomicaYSeno = new Suma(new Polinomica([0, 0, 0, 2]), seno);
    // f(x) = sin(x^2)
    let senoDelCuadrado = new Composicion(seno, cuadrado);

    console.log(constantePorSeno.evaluar(Math.PI/2)); // 3
    console.log(senoCuadrado.evaluar(Math.PI/4)) // 0.5
```

```
    console.log(constante3.evaluar(5), constante3.evaluar(3),
constante3.evaluar(16)) // 3 3 3
    console.log(cuadrado.evaluar(2)); // 4
    console.log(coseno.evaluar(Math.PI)); // -1
    console.log(cosenoCuadrado.evaluar(Math.PI/6)); // 0.25
    console.log(polinomicaYSeno.evaluar(1)); // 2.84147...
    console.log(senoDelCuadrado.evaluar(1)); // 0.7086...
}

main();
```

Si bien el resultado esperado es 0.5 cuando evaluamos el seno al cuadrado de $\pi/4$, el programa nos devuelve un valor cercano pero no exactamente 0.5 debido a la naturaleza de la aritmética de punto flotante utilizada por las computadoras para representar números reales.
Es cierto que existen técnicas para manejar mejor este problema, como la representación de números decimales mediante fracciones o la aplicación de técnicas de redondeo. Sin embargo, estos temas pueden ser más avanzados y no están dentro del alcance de este curso.

En cambio, es importante tener en cuenta las limitaciones de la aritmética de punto flotante al trabajar con números reales en programas y estar preparados para enfrentar problemas similares en el futuro.

Generics

Introducción

En la programación, Generics se refiere a una forma de escribir código que es flexible y puede trabajar con diferentes tipos de datos. En lenguajes como TypeScript, esto se logra mediante el uso de variables de tipo que representan tipos genéricos, que pueden ser definidos cuando se utiliza el código. Los Generics pueden ser utilizados en interfaces, clases y funciones para crear código más versátil y reutilizable. Al usar Generics, podemos escribir código que pueda manejar una amplia gama de tipos de datos sin tener que escribir código separado para cada tipo.

Generics en Clases

Tenemos lo que se llama listas, que son similares a los arreglos pero sus elementos no están indexados, sino que están compuestos de nodos que están conectados entre sí de diferentes maneras. Dependiendo de la forma en que se conectan los nodos, se obtienen comportamientos diferentes para la lista.

Un ejemplo claro de uso sería un cajero en el supermercado. Los clientes se colocan en una cola en el supermercado y se les atiende de uno en uno. Para modelar esta cola o lista encolada, necesitaríamos establecer el formato general de cómo va a estar compuesta la misma. Cada cliente, o nodo, estaría compuesto por él mismo y apuntaría al siguiente nodo:

```
interface INodo<T> {
    elemento: T;
    siguiente?: INodo<T>;
}

interface ICola<T> {
    primerElemento?: INodo<T>;
}
```

En el código, podemos notar que hay una letra T donde tienen que estar los tipos de las variables. Esta T se llama genérica, y es literalmente una variable con un tipo de datos genérico. Lo bueno de esto es que si queremos una lista de cualquier tipo de datos, lo único que tendremos que hacer es definir este tipo de datos cuando declaramos una variable.

Ahora, ¿cómo podemos llenar esa lista y cómo podemos imprimirla? Simple. Hay que definir diferentes métodos y crear una clase que los implemente:

```
interface INodo<T> {
    elemento: T;
    siguiente?: INodo<T>;
}

interface ICola<T> {
    primerElemento?: INodo<T>;
    adicionarElemento: (elemento: T) => ICola<T>;
    imprimirCola: () => void;
}

class Cola<T> implements ICola<T> {
    primerElemento?: INodo<T>;

    adicionarElemento(elemento: T): Cola<T> {
        let nodoActual = this.primerElemento;

        if( !nodoActual ) {
            this.primerElemento = {
                elemento: elemento,
                siguiente: this.primerElemento
            };
        } else {
            while( !!nodoActual ) {
                if( !nodoActual.siguiente) {
                    nodoActual.siguiente = {
                        elemento: elemento,
                        siguiente: nodoActual.siguiente
                    };
                    nodoActual = nodoActual.siguiente.siguiente;
                } else {
                    nodoActual = nodoActual.siguiente;
                }
            }
        }
        return this;
    }

    imprimirCola(): void {
        let nodoActual = this.primerElemento;
        while( !!nodoActual ) {
            console.log(nodoActual.elemento)
            nodoActual = nodoActual.siguiente;
```

```
        }
    }
}
```

También, podríamos hacer que al adicionar un elemento a la cola, esta se devuelva a sí misma y, de esta forma, poder añadir varios elementos a la vez. Para probar que nuestra estructura funciona adecuadamente, podemos crear una cola de números dentro de una función llamada testCola:

```
import Cola from "./Clases/Cola";

const testCola = () => {
    const colaDeNumeros = new Cola<number>();
    colaDeNumeros.adicionarElemento(1).adicionarElemento(2)
                    .adicionarElemento(27).adicionarElemento(40);
    colaDeNumeros.imprimirCola(); // 1 2 27 40
}

testCola();
```

Se imprimirán en pantalla los números en el orden en que fueron añadidos (1, 2, 27, 40).

Lo interesante de todo esto es que podemos usar nuestra Cola con distintos tipos de datos y en todos los casos funcionará. Este es el poder de los tipos de datos genéricos. Sirve tanto para interfaces, como clases, como también funciones.

Por último, podemos modificar los métodos para imprimir la cola y para adicionar un elemento para cambiar su funcionalidad. Por ejemplo, si se quiere una pila, el último elemento que se adiciona debe ser el primero en ser recorrido.

Generics en funciones

Los Generics en TypeScript permiten además, crear funciones que puedan trabajar con diferentes tipos de datos, sin conocer previamente qué tipo específico se va a utilizar en tiempo de compilación.

Supongamos que queremos escribir una función que tome un arreglo de elementos y devuelva el primer elemento de la lista. Podríamos hacerlo de la siguiente manera:

```
function primerElemento(arr: any[]): any
{
  return arr[0];
```

```
}
```

Sin embargo, esta función tiene un problema: si pasamos una lista que no contiene ningún elemento, la función arrojará un error que no fue previamente tomado en cuenta al hacer la función. Además, esta función sólo acepta listas de tipo any, lo que significa que no se puede garantizar el tipo de los elementos de la lista.

Para solucionar esto, podemos utilizar Generics en la función. En este caso, queremos que la función funcione con cualquier tipo de arreglo, por lo que crearemos un parámetro genérico T que represente el tipo de datos del arreglo. También, introduciremos cómo arrojar errores por cuenta propia, para mejorar los errores que se muestran en pantalla.

Para arrojar errores, se usa la palabra reservada *throw*, que enviará un error al programa y va a proporcionar un mensaje descriptivo indicando que la lista está vacía. Esto es útil para informar al usuario del error y detener la ejecución de la función, de ser necesario

La función nos quedaría:

```
function primerElemento<T>(arr: T[]): T {
  if (arr.length === 0) {
    throw new Error("La lista está vacía");
  }
  return arr[0];
}
```

Ahora, esta función aceptará listas de cualquier tipo de datos y devolverá el primer elemento de la lista con el mismo tipo de datos que los elementos de la lista. Si pasamos una lista vacía, la función arrojará un error. Podemos llamar a la función con diferentes tipos de listas, como una lista de números, una lista de cadenas de texto, una lista de objetos personalizados, entre otros:

```
import primerElemento from "./Funciones/primerElemento";

const numeros = [1, 2, 3, 4];
const primerNumero = primerElemento(numeros );
console.log(primerNumero) // devuelve 1

const cadenas = ["hola", "mundo"];
const primerCadena = primerElemento(cadenas );
```

```
console.log(primerCadena) // devuelve "hola"

interface Persona {
    nombre: string;
    edad: number;
}

const personas: Persona[] = [
    { nombre: "Juan", edad: 25 },
    { nombre: "María", edad: 30 },
];


const primerPersona = primerElemento(personas);
console.log(primerPersona) // devuelve { nombre: "Juan", edad: 25 }
```

En resumen, los Generics permiten escribir funciones que sean más flexibles y reutilizables al aceptar diferentes tipos de datos y proporcionar un tipo de retorno seguro.

Recursividad

Introducción

La recursividad es una técnica poderosa en programación, que permite resolver problemas de manera elegante y eficiente. En este curso, vamos a ver cómo usar la recursividad en TypeScript para resolver problemas de programación.

La recursividad es una técnica en la que una función se llama a sí misma para resolver un problema. En lugar de resolver un problema de manera iterativa, la recursividad permite descomponer el problema en subproblemas más pequeños y resolverlos de manera recursiva.

La recursividad se utiliza comúnmente para resolver problemas que se pueden dividir en subproblemas similares más pequeños. En estos casos, la recursividad a menudo es más clara y más fácil de entender que las soluciones iterativas.

Ejemplo: Sumar primeros N números

Un ejemplo común de la utilización de la recursividad es la suma de los primeros N números enteros. Podemos resolver este problema de manera recursiva de la siguiente manera:

```
function sumaPrimerosN(n: number): number {
  if (n === 0) {
    return 0;
  } else {
    return n + sumaPrimerosN(n - 1);
  }
}

console.log(sumaPrimerosN(5)); // 15
```

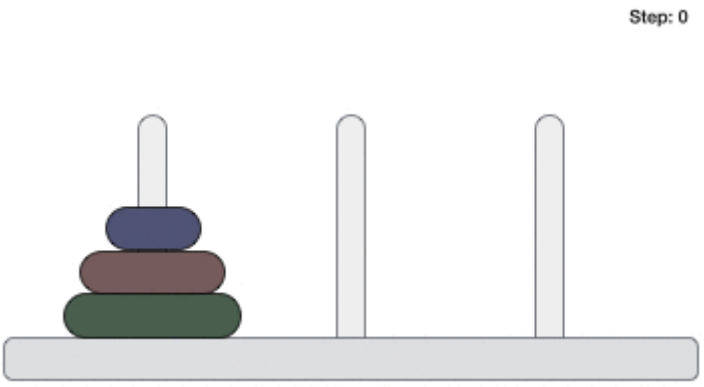
En este ejemplo, utilizamos la recursividad para descomponer el problema de suma de N números enteros en un subproblema más pequeño: la suma de los primeros N-1 números enteros. Resolvemos este subproblema recursivamente utilizando la función sumaPrimerosN hasta que llegamos al caso base en el que n es 0.

Torres de Hanoi

Hay tres palos parados y en uno de ellos hay una pila de discos de distintos tamaños, cada uno encima del otro en orden de grandeza. El objetivo es pasar toda la pila de discos al tercer palo, pero se puede mover un solo disco a la vez y nunca se puede poner uno más grande encima de uno más chico.

Para resolver este problema, usamos un algoritmo recursivo que hace lo siguiente:

1. Mover los discos más pequeños (todos menos el más grande) del primer palo al segundo palo, usando el tercer palo como ayudante.
2. Mover el disco más grande del primer palo al tercer palo.
3. Mover los discos que estaban en el segundo palo al tercer palo, usando el primer palo como ayudante.
4. Repetimos estos pasos para cada grupo de discos que quedan en el primer palo, hasta que pasemos toda la pila al tercer palo.



Vamos a tener en cuenta que los discos estarán identificados por un número, siendo 1 el disco más chico, y N el disco más grande (habiendo N discos)

Para redactar este problema de otra forma, deberíamos dividirlo en caso base, primer problema y segundo problema.

Al revés de lo que algunos piensan, nuestro caso base va a ser el disco más pequeño, que será el que estará libre de moverse a dónde quiera. Debido a esto, se mueve siempre desde donde está hacia el objetivo que le indican.

Si el disco no es el más pequeño, tendrá que mover primero a todos los demás discos arriba de él (discos con números más chicos) al palo auxiliar. Después, va a poder moverse él hacia el objetivo.

Por último, va a mover a los demás discos que estaban encima de él (discos con números más grandes) hacia el palo objetivo.

Nuestro código termina quedando de la siguiente manera:

```
function torresDeHanoi(
  n: number,
  desde: string,
  hacia: string,
  auxiliar: string
) {
  if (n === 1) {
    console.log(`Mover disco 1 desde ${desde} hacia ${hacia}`);
  } else {
    torresDeHanoi(n - 1, desde, auxiliar, hacia);
    console.log(`Mover disco ${n} desde ${desde} hacia ${hacia}`);
    torresDeHanoi(n - 1, auxiliar, hacia, desde);
  }
}

// Ejemplo de uso:
torresDeHanoi(3, "A", "B", "C");
```

Estructura general de recursividad

Ahora, en general, un código recursivo tiene la siguiente estructura:

```
function funcionRecursiva(parametros): tipoDeRetorno {
  if (/* condición de caso base */) {
    // Valor de retorno del caso base
  } else {
    // Descomponer el problema en subproblemas más pequeños
    const problemaChico1 = /* descomposición del problema 1*/;
    .
```

```
    .
    .
    const problemaChicoN = /* descomposición del problema N*/;
    // Resolver subproblemas
    const resultado1 = funcionRecursiva(problemaChico1);
    .
    .
    .
    const resultadoN = funcionRecursiva(problemaChicoN);
    // Combinar los resultados de los subproblemas
    const result = /* combinación de resultados */;
    return result;
  }
}
```

En la estructura general de una función recursiva en TypeScript, tenemos un caso base que se encarga de detener la recursión y proporcionar un valor de retorno. Si no estamos en el caso base, descomponemos el problema en subproblemas más pequeños, resolvemos estos subproblemas recursivamente y luego combinamos los resultados de los subproblemas para obtener el resultado final. Finalmente, devolvemos el valor de retorno de la función recursiva.

Excepciones

Introducción

A esta altura del curso, es probable que muchos de los lectores hayan enfrentado algún problema mientras ejecutaban su programa. Que el programa nunca se corte también es una excepción en sí misma. Esta gran problemática, en realidad, es una solución para poder separar lo que debería ocurrir normalmente de algún error o valor excepcional del programa.

Ejemplo de manejo de excepciones

Supongamos un error simple. Quiero acceder a las propiedades de un objeto que no está definido aún (como una posición de un arreglo no iniciado, por ejemplo):

```
let numeros: number[] = [];
numeros[5].toFixed(); // TypeError: Cannot read properties of undefined
(reading 'toFixed')
```

Como se puede ver, la posición 5 del arreglo de objetos aún no existe, por lo que no se puede utilizar un método en ese objeto inexistente. Por lo tanto, al ejecutar este código, el compilador de TypeScript arrojará un error.

Para manejar esta situación y mostrar un mensaje de confirmación al usuario de que el código se ejecutó correctamente o no, se utilizan bloques try {... } catch. En la sección try, se coloca el código que puede generar una excepción, y en la sección catch, se proporcionan instrucciones para manejar el error si se produce una excepción:

```
let numeros: number[] = [];

try{
    numeros[5].toFixed();
} catch( e: any ) { // Solo puede ser del tipo any o unknown. Si no se
conoce a qué clase pertenece el error, dejar en any
    console.log("Se produjo un error: " + e.message);
}

console.log("El programa terminó de ejecutarse.");
```

Con este código, el programa terminará exitosamente al atrapar la excepción y mostrar un mensaje al usuario en caso de error. Además, se imprimirá un mensaje al final del

programa para indicar que el programa se sigue ejecutando después de haber atrapado la excepción.

Tipos de Error

También, se pueden crear tipos de Error, que son clases que extienden de la misma. En ese caso, se puede configurar más información acerca del error o excepción

Para ello, primero tendremos que instalar el tipo @types/node, el cual tendrá la lógica de setPrototypeOf (el package.json se creará automáticamente):

```
$ npm i --save-dev prompt @types/prompt @types/node
```

```
export default class ArgumentoNoEsNumeroError extends Error {
    constructor( ) {
        super('El argumento ingresado debe ser un número');
        Object.setPrototypeOf(this, ArgumentoNoEsNumeroError.prototype);
// Le indicamos a la instancia que es del tipo error

// Esto pasa debido a cómo Typescript maneja el constructor
    }
}
```

```
export default class NumeroEsNegativoError extends Error {
    constructor( ) {
        super('El argumento ingresado debe ser un número positivo');
        Object.setPrototypeOf(this, NumeroEsNegativoError.prototype);
// Le indicamos a la instancia que es del tipo error
// Esto pasa debido a cómo Typescript maneja el constructor
    }
}
```

Interacción con el usuario y manejo de excepciones

Claramente, no tiene sentido arrojar excepciones sin una razón válida, sino que probablemente se produzcan por la interacción de nuestro programa con el usuario o con archivos que se lean. Para probar la respuesta del programa al usuario, aprenderemos sobre la función prompt(), de la librería prompt-sync, que tomará el input del usuario a través del teclado.

Para ello, primero tendremos que instalar la dependencia prompt-sync, junto con su tipo:

```
$ npm i --save-dev prompt @types/prompt prompt-sync
```

Analizaremos el siguiente código línea por línea:

```
import Polinomica from "../Funciones/Clases/Polinomica";
import Funcion from "../Funciones/Interfaces/Funcion";
import ArgumentoNoEsNumeroError from "./ArgumentosNoEsNumeroError";
import NumeroEsNegativoError from "./NumeroEsNegativoError";
const prompt = require("prompt-sync")({ sigint: true });



const leerNumero = (): number | never => {
    const nuevoNumero = parseFloat(prompt('Por favor, ingrese un número
para evaluar la función: '));
    if( isNaN(nuevoNumero) ) {
        throw new ArgumentoNoEsNumeroError;
    }
    if (nuevoNumero < 0) {
        throw new NumeroEsNegativoError;
    }
    return nuevoNumero;
}

const main = (): void => {
    let funcionPoderosa: Funcion = new Polinomica([0,0,1]) // f(x) = x^2
    let numeros: number[] = [];
    let nuevoNumero: number;
    while( numeros.length < 5 ) {
        try {
            nuevoNumero = leerNumero();
            numeros.push(nuevoNumero);
        } catch (e: any) {
            if( e instanceof  ArgumentoNoEsNumeroError ) {
                console.log('Ha ingresado un valor incorrecto.');
            }
            if(e instanceof NumeroEsNegativoError) {
                console.log('Ha ingresado un valor erróneo.');
            }
        }
    }
```

```
    for( let numero of numeros) {
        console.log(funcionPoderosa.evaluar(numero));
    }
}

main();
```

En primer lugar, importamos la librería prompt-sync y la iniciamos con la opción sigint en true. Esto es para que podamos interrumpir la solicitud de input con la combinación Ctrl + C.

Luego, definimos una función llamada leerNumero, donde pediremos al usuario que ingrese un número y, si este es negativo o si el usuario ingresa cualquier cosa que no sea un número, se lanzará un error. El tipo del error se especifica como never.

Dentro de la función main, definimos una función de ejemplo (en este caso, un polinomio) junto con un arreglo de números y una variable para almacenar los nuevos números que se agreguen.
Dentro de un ciclo while, que se ejecutará hasta que la matriz tenga cinco elementos, intentaremos leer un número. Si la operación tiene éxito, agregaremos el número al arreglo. Si se produce un error, se imprimirá un mensaje en la pantalla indicando que se produjo un error y qué tipo de error fue.

Una vez que se han leído los cinco números, el programa procederá a imprimir la función evaluada en esos 5 puntos.

Lectura de Archivos

Introducción

A veces necesitamos obtener información de otros programas o guardarla para accederla más tarde o dársela a otro programa. Esto se puede hacer a través de las bases de datos. Hay varias formas de representar una base de datos, y hay motores de bases de datos muy buenos, como PostgreSQL, Neo4J, Mongodb, etc., aunque no exploraremos estos en este curso debido a su duración.

Las bases de datos más comunes probablemente sean las de archivos con un formato específico. Por ejemplo, los archivos JSON se pueden guardar para funcionar como bases de datos. Una representación común de la información es mostrarla en tablas.

Async

Una función asincrónica es aquella que puede ejecutar tareas que llevan tiempo (como hacer una solicitud a un servidor o leer un archivo del disco) sin bloquear el hilo principal de ejecución del programa. En lugar de eso, la función asincrónica continúa ejecutándose mientras se espera que se completen estas tareas en segundo plano.

Cuando una función se define como *async*, automáticamente se convierte en una función que devuelve una promesa. La promesa se resuelve con el valor devuelto por la función, o se rechaza con un error si ocurre alguno.

Para llamar a una función asincrónica, se utiliza la palabra clave await. await se utiliza para esperar a que una promesa se resuelva antes de continuar con la ejecución del programa. Cuando se llama a una función asincrónica con await, el programa espera a que la promesa devuelta por la función se resuelva antes de continuar ejecutando el código siguiente.

En resumen, async y await son palabras clave de JavaScript que se utilizan para trabajar con funciones asincrónicas y promesas. Con async se define una función asincrónica que devuelve una promesa, y con await se espera a que una promesa se resuelva antes de continuar con la ejecución del programa.

Un corto ejemplo de esto sería:

```
function esperar(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
}
```

```
async function ejemploAsincronico(): void {
    console.log("Inicio de la función");

    // Esperamos 1 segundo antes de continuar con la ejecución
    await esperar(1000);

    console.log("Fin de la función");
}

ejemploAsincronico();
```

setTimeout, lo que hace es esperar la cantidad de milisegundos especificada e invocar a la función que se le pasa como primer parámetro. En este caso, esa función es *resolve*, que es como se resuelve una promesa (*Promise*).

En este caso vamos a ver que nos imprimirá en pantalla “Inicio de la función”, y 1 segundo después, nos imprimirá “Fin de la función”.

Si, en cambio, no le ponemos *await* a esperar(1000), nos debería imprimir “Fin de la función” casi un instante después de que imprimiera “Inicio de la función”

```
function esperar(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
}

async function ejemploAsincronico(): void {
    console.log("Inicio de la función");

    // Esperamos 1 segundo antes de continuar con la ejecución
    esperar(1000);

    console.log("Fin de la función");
}

ejemploAsincronico();
```

Ejemplo de lectura y consultas rápidas con Danfo.js

Si queremos manejar archivos de bases de datos, podemos utilizar Danfo.js, una librería que maneja estructuras de datos similares a pandas en Python (otro lenguaje de programación). Para ello, instalaremos la dependencia de la siguiente manera:

```
npm i danfojs-node
```

Nota: Requiere tener instalado Visual Studio de Microsoft (incluyendo Desktop development with C++). Si no está instalado, va a arrojar un error.

Para la parte de lectura de archivos, utilizaremos el siguiente archivo de ejemplo.

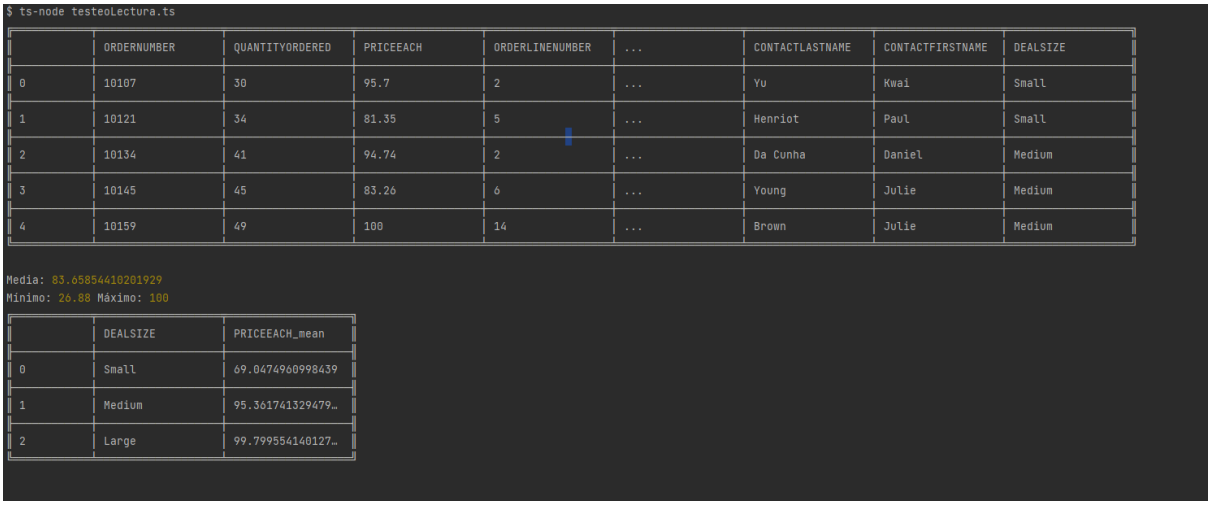
Ahora, crearemos un programa que abra este archivo e imprima los datos. Es tan simple como indicarle a Danfo que abra el archivo ubicado en la dirección indicada, y este se encargará de levantar el conjunto de datos correctamente. Luego, podemos hacer consultas a los datos de manera rápida:

```
import {readCSV} from 'danfojs-node';
import { DataFrame } from 'danfojs-node/dist/danfojs-base';

async function main() {
    let df: DataFrame;
    try {
        df = await readCSV('sales_data_sample.csv')
    } catch (e) {
        console.log('Ocurrió un error leyendo el archivo. Saliendo')
        return;
    }
    if (!!df) {
        df.head().print();
        // Imprime los primeros valores del archivo en formato de tabla.
        console.log('Media:', df.PRICEEACH.mean())
        // Accede a la columna PRICEEACH y encuentra la media.
        console.log('Mínimo:', df.PRICEEACH.min(), 'Máximo:',
df.PRICEEACH.max())
        // Accede a la columna PRICEEACH y encuentra el mínimo y el
máximo.
        let groupbyDealSize = df.groupby(['DEALSIZE'])
        // Agrupa por DEALSIZE
        groupbyDealSize.agg({'PRICEEACH': 'mean'}).print();
        // Imprime la media del PRICEEACH, agrupando por DEALSIZE
    }
}

main();
```

El DataFrame es un tipo de datos creado por Danfo, que representa una estructura de datos bidimensional. Esto significa que se ve como una tabla. *readCSV* es la función encargada de leer la tabla que le pasamos, en formato CSV.

Para obtener los valores de una columna, simplemente colocamos el nombre de la columna después de llamar al conjunto de datos. Si queremos obtener el promedio de cierta columna, podemos utilizar la función mean, que también puede ser utilizada en todo el conjunto de datos. Si deseamos obtener el máximo o el mínimo de una columna, podemos usar las funciones min y max, respectivamente, que también se pueden usar en todo el conjunto de datos. Si queremos agrupar por alguna columna de interés, usamos la función groupby y le pasamos un arreglo con los nombres de las columnas de interés. Si deseamos aplicar alguna función "resumen" de alguna columna o aplicar una función sobre varias columnas después de agrupar, usamos la función agg (de función agregativa).

El resultado final del programa es el siguiente:



Para más consultas posibles a hacer al dataset, se verán herramientas en la próxima clase. De todas formas, la siguiente página es una buena introducción a este mundo, para el que quiere saber más.