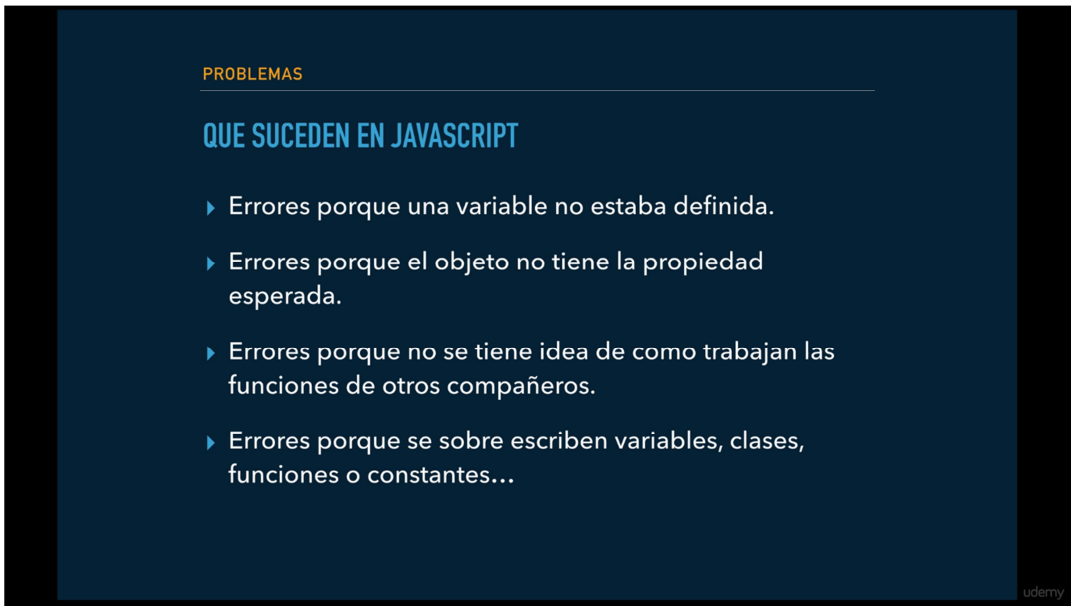
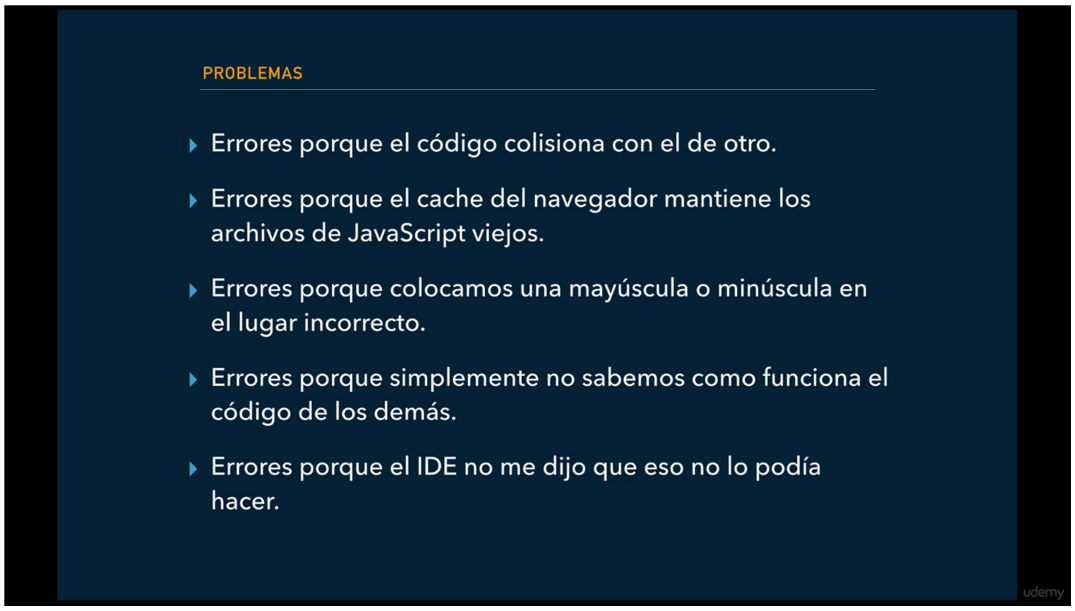
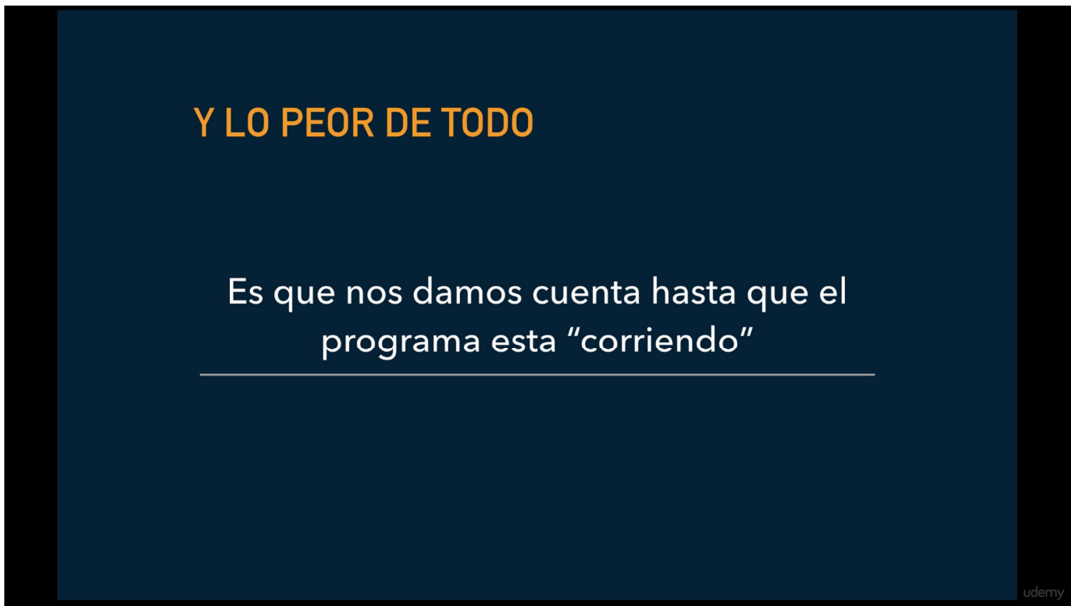
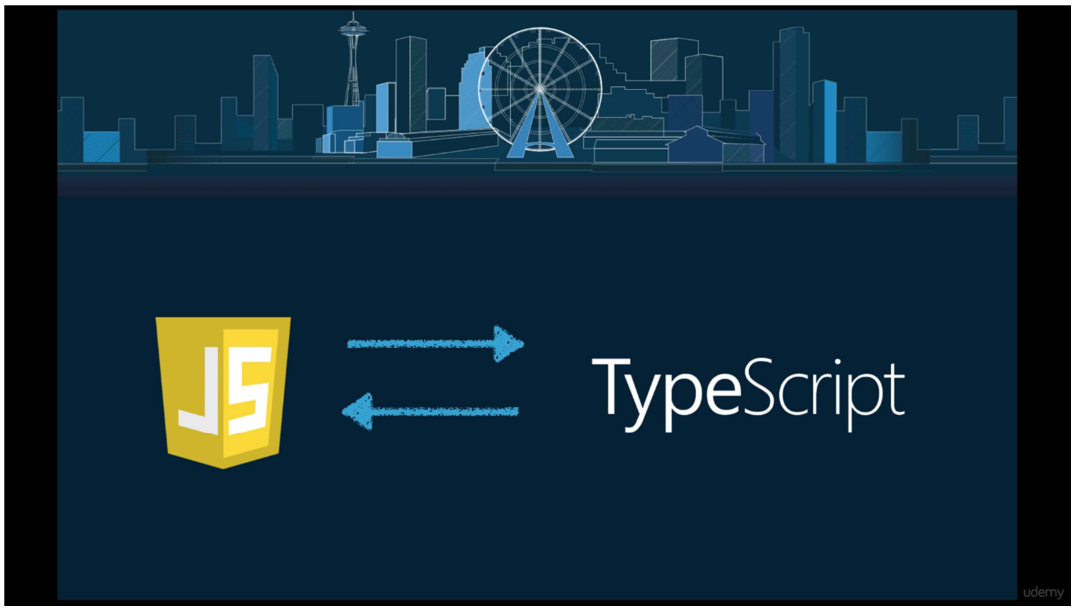
Type Script

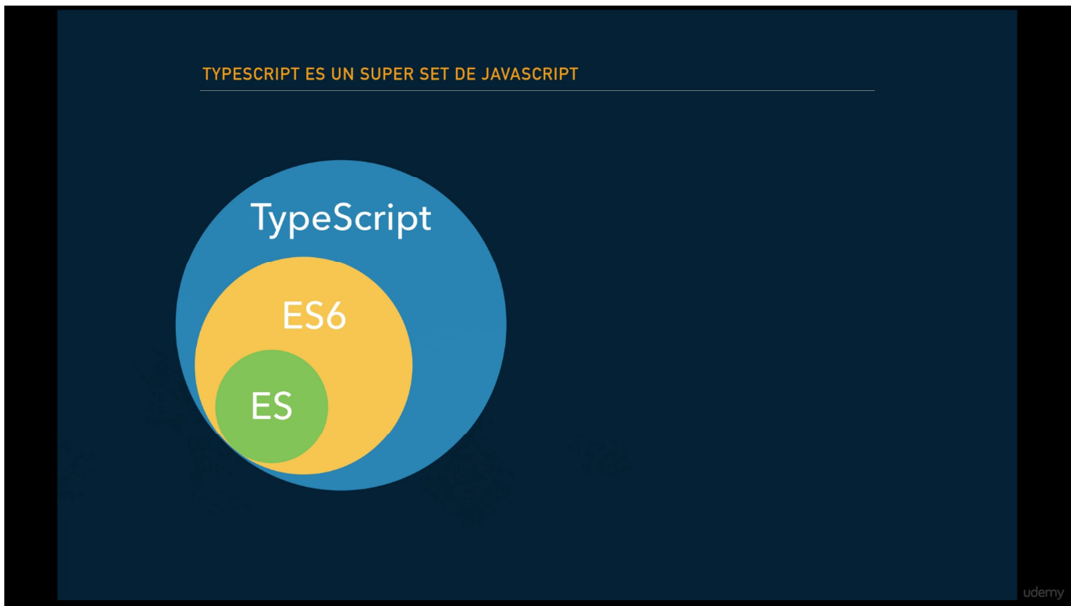

JAVASCRIPT CARECE DE MUCHAS COSAS
CARACTERISTICAS FALTANTES
Tipos de variables
Errores en tiempo de escritura
Auto completación dependiendo de la variable
Método estático de programación
Clases y módulos (antes de ES6)
Entre otras cosas...
udemy


PROBLEMAS
QUE SUCEDEN EN JAVASCRIPT
Errores porque una variable no estaba definida.
Errores porque el objeto no tiene la propiedad esperada.
Errores porque no se tiene idea de como trabajan las funciones de otros compañeros.
Errores porque se sobre escriben variables, clases, funciones o constantes...
udemy


PROBLEMAS
Errores porque el código colisiona con el de otro.
Errores porque el cache del navegador mantiene los archivos de JavaScript viejos.
Errores porque colocamos una mayúscula o minúscula en el lugar incorrecto.
Errores porque simplemente no sabemos como funciona el código de los demás.
Errores porque el IDE no me dijo que eso no lo podía hacer.
udemy


Y LO PEOR DE TODO
Es que nos damos cuenta hasta que el programa esta "corriendo"
udemy





ECMAScript

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto como estándar por Netscape Communications Corporation. Actualmente está aceptado como el estándar ISO 16262.

ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases.

La mayoría de navegadores de Internet incluyen una implementación del estándar ECMAScript, al igual que un acceso al Document Object Model para manipular páginas web. JavaScript está implementado en la mayoría de navegadores, Internet Explorer de Microsoft usa JScript. El navegador Opera tenía su propio

intérprete de ECMAScript con extensiones para soportar algunas características de JavaScript y JScript, actualmente Opera está basado en Chromium (y utiliza su intérprete). Cada navegador tiene extensiones propias al estándar ECMAScript, pero cualquier código que se adecúe al estándar debería funcionar en todos ellos.

ActionScript, para Adobe Flash, también está basado en el estándar ECMAScript, con mejoras que permiten mover, crear y analizar dinámicamente objetos, mientras la película está en ejecución.

Abrir terminal de comandos

Compilar archivo ts -> genera js

tsc archivots.ts o tsc archivos

Modo observador

tsc archivots –w

Salir Modo Observador

Ctrl + C

Crear proyecto type script

tsc -init

ejecutar tsc, ahora debe compilar la totalidad de archivos ts de la carpeta generando los correspondientes js

var vs let (para verificar diferencia ver el js resultante)

```
var mje = "hola";
if(true){
    var mje="adios";
}
console.log(mje);
```

```
let mje = "hola";
if(true){
    let mje="adios";
}
console.log(mje);
```

Constantes

```
const DEBUG = false;
```

Tipos de Datos

```
let cadena:string = "PEPE";
let numero:number = 123;
let varbool:boolean = true;
let hoy:Date = new Date();

let cualquiera:any; //acepta cualquier tipo como en js

let cliente = {
    nombre:"juan",
    edad:23
}
```

Uso de ${}

```


let nombre:string = "Juan";
let apellido:string = "Castro";
let edad:number = 25;


let texto = "Hola " + nombre + " " + apellido + " (" + edad + ")";

console.log(texto);

//usar apostrofe invertido
let texto2 = `Hola ${nombre} ${apellido} (${edad})`;

console.log(texto2);


console.log(`${5 + 6}`);
```

Funciones: Parámetros opcionales, obligatorios y por defecto

```
function metodoX(obligatorio:string, porDefecto:string = "valor asignado",
optativo?:string){

    if(optativo){
        console.log(`${obligatorio} ${porDefecto} ${optativo}`);
    }else{
        console.log(`${obligatorio} ${porDefecto}`);
    }

}

metodoX("Funciona");
```

Funciones Flecha ES6

Las funciones de flecha son una forma reducida de declarar una función TS/JS

La sintaxis base seria ***(parametros) => retorno***

```
//ejemplo 1
let mifuncion = function (a) {
    return a;
};
let mifuncionFlecha = (a) => a;

console.log(mifuncion("Normal"));
console.log(mifuncionFlecha("De Flecha"));
```

```
//ejemplo 2
let mifuncion2 = function (a:number, b:number) {
    return a + b;
};
let mifuncionFlecha2 = (a:number, b:number) => a + b;
console.log(mifuncion2(5, 7));
console.log(mifuncionFlecha2(5, 7));
```

```
//ejemplo3
let mifuncion3 = function (nombre:string) {
      nombre = nombre.toUpperCase();
      return nombre;
};
let mifuncionFlecha3 = (nombre:string) =>{
      nombre = nombre.toUpperCase();
      return nombre;
};
console.log(mifuncion3(“pepe”));
console.log(mifuncionFlecha3(“pepe”));
```

Los 3 ejemplos anteriores muestran cómo se implementa una función flecha y como se define la sintaxis, sin embargo hay una característica diferencial de la función flecha y es la referencia al objeto **this**

```
//ejemplo 4 - caso diferencial
let tenista = {
    nombre: "Federer",
    smash: function () {
        console.log(this.nombre + " smash!!");
```

```
    }
};
tenista.smash();
```

Imprime por consola: Federer Smash!!

Qué pasa si agregamos un tiempo de espera en la función de esta forma:

```
//pauso ejecución
let tenista = {
    nombre: "Federer",
    smash: function () {
        setTimeout(function() {
             console.log(this.nombre + " smash!!");//this apunta al objeto global
window
        }, 1500);

    }
};
tenista.smash();
```

Imprime por consola: undefined Smash!!

Esto es así porque el **this** está apuntando al objeto Global en este caso Windows y no al objeto tenista. Esto lo soluciono mediante la función Flecha.

```
//pauso ejecución usando flecha
let tenista = {
    nombre: "Federer",
    smash() {
        setTimeout(() => console.log(this.nombre + " smash!!"), 1500);//this no es
afectado al ejecutarse dentro de otra función
    }
};
tenista.smash();
```

Imprime por consola: Federer Smash!!

Al hacer uso de la función flecha **this** apunta al objeto que lo contiene.

Destructuración de Objetos y Arreglos

```
let pais = {
    nombrePais:"Argentina",
    superficie:"2.78 Km2",
    poblacion:44000000
}

let nomPais = pais.nombrePais;
```

```
let supPais = pais.superficie;
let pobPais = pais.poblacion;

console.log(nomPais, supPais, pobPais );

//destructuración

let { nombrePais, superficie, poblacion} = pais;

console.log(nombrePais, superficie, poblacion );

//ARREGLOS
let paises:string[] = ["Argentina","Peru","Uruguay"];

let [pais1, pais2, pais3] = paises;


console.log(pais1, pais2, pais3);
```

Promesas en ES6

Las promesas se utilizan para la realización de tareas asíncronas, como por ejemplo la obtención de respuesta a una petición HTTP.

Una promesa puede tener 4 estados:

- **Pendiente**: Es su estado inicial, no se ha cumplido ni rechazado.
- **Cumplida**: La promesa se ha resuelto satisfactoriamente.
- **Rechazada**: La promesa se ha completado con un error.
- **Arreglada**: La promesa ya no está pendiente. O bien se ha cumplido, o bien se ha rechazado.

Sintaxis

new Promise(function(resolve, reject) { ... });

Cuando creamos una promise, le pasamos una función en cuyo interior deberían producirse las operaciones asíncronas, que recibe 2 argumentos:

- **Resolve**: Es la función que llamaremos si queremos resolver satisfactoriamente la promesa.
- **Reject**: Es la función que llamaremos si queremos rechazar la promesa.

```
let promise = new Promise(function(resolve, reject){
 setTimeout(()=> {
        console.log("Promesa finalizada");
        //sale bien
        resolve();
        //sale mal
        reject();
```

```
    }, 1500)
});

promise.then(
  function(value){
    console.log("ejecutar si todo sale bien");
  },
  function(reason){
    console.log("ejecutar si error ", reason);
  }
);

promise.then(function(success){console.log("another callback")});


console.log("PASO 1");
```

Interfaces en TypeScript

Define una estructura que debe respetarse para que compile correctamente.

```
interface Soldado{
    nombre:string;
    tipo:string;
}

function enviarMision(soldado:Soldado){
    console.log("Enviar a Mision a " + soldado.nombre);
}

function enviarCuartel(soldado:Soldado){
    console.log("Enviar al Cuartel a " + soldado.nombre);
}

let ryan:Soldado = {
    nombre: "ryan",
    tipo:"infanteria"
};

enviarMision(ryan);
enviarCuartel(ryan);
```

Clase Basica

```
class Vehiculo {

    marca:string;
    modelo:string;
    patente:string;
    numeroSerie:number;

   //constructor por defecto
    /*constructor(){
        console.log("ejecuta constructor de clase");

    }*/
    //constructor sobrecargado
    constructor(marca:string, modelo:string, patente:string, numeroSerie:number){
        console.log("ejecuta constructor sobrecargado de clase");
        this.marca = marca;
        this.modelo = modelo;
        this.patente = patente;
        this.numeroSerie = numeroSerie;
    }

    acelerar(velocidad:number){
        console.log("Acelerar vehiculo a " + velocidad + " km/hora");
    }
}

//let auto:Vehiculo = new Vehiculo();
let auto:Vehiculo = new Vehiculo("Ford", "Ka", "AGD345RD", 1596);
auto.acelerar(120);
```

Exportar e Importar Clases

Para poder hacer visible una clase ts la misma debe estar declarada con la palabra reservada export

```
export class Vehículo{}
```

Si queremos importar una clase para hacer uso de ella debemos indicarlo mediante la clausula:

```
import {Vehiculo} from "./claseBasica";

let auto:Vehiculo = new Vehiculo("Ford", "Ka", "AGD345RD", 1596);
auto.acelerar(120);
```

Siendo claseBasica el nombre del archivo ts que contiene a la clase Vehículo