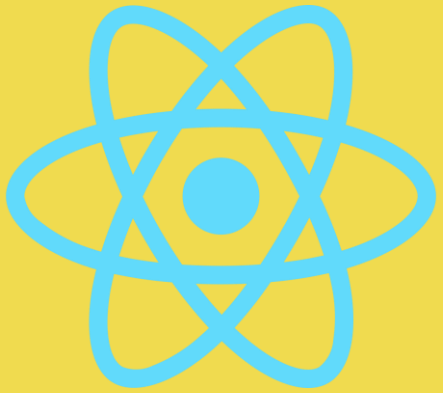
Programación Estructurada con TypeScript

Qué pasa con JavaScript?

JavaScript no fue diseñado para soportar proyectos de mediano y gran tamaño por lo que tiene un “tipado débil”, lo que complica el desarrollo de proyectos en JS puro.

Por eso existen los Frameworks y herramientas JS



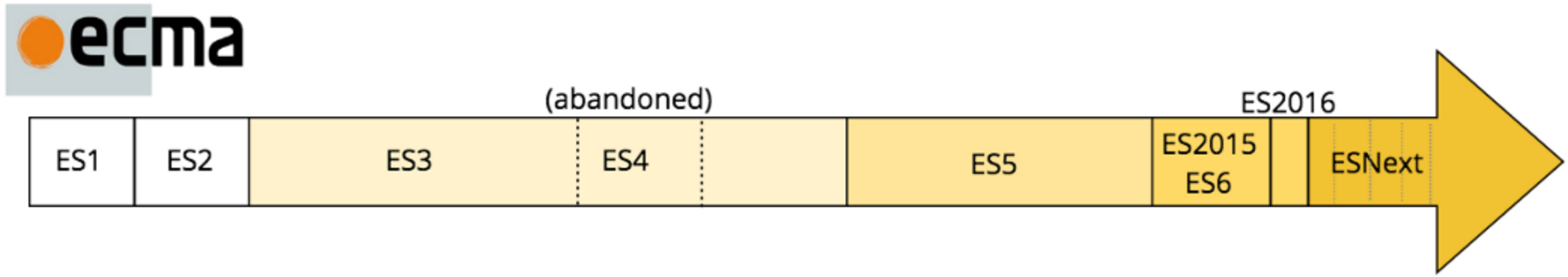
Entonces que es TypeScript?

Es un superset de JavaScript de “tipado fuerte” que transpila código TypeScript en código ECMAScript estándar.

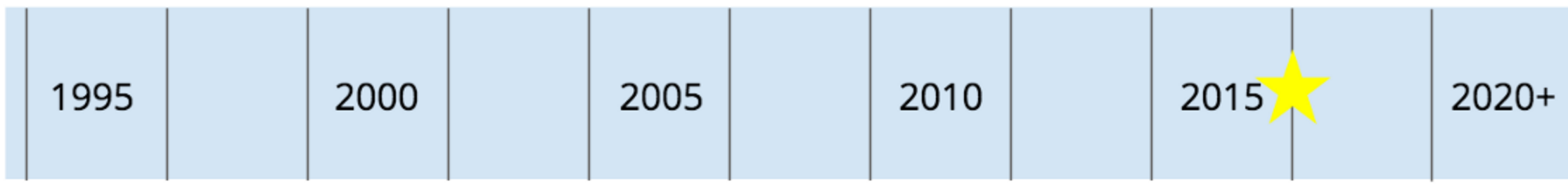

Oye oye mas despacio
cerebrito

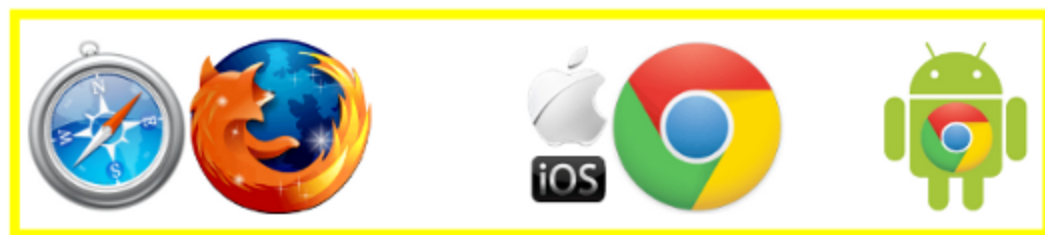
- **Superset**: contiene todas las versiones de ES para JS por lo cual podremos usar utilidades de las ultimas versiones de ES sobre versiones pasadas.

- **Tranpilador**: que convierte código TypeScript en ES standart (ES 5-6).


ecma
(abandoned)
ES2016
ES1
ES2
ES3
ES4
ES5
ES2015
ES6
ESNext


JS


1995
2000
2005
2010
2015
2020+


iOS

Trabajando con TypeScript

- **Iniciar un nuevo proyecto**:

  ```
  tsc –init
  ```

- **Transpilar automáticamente**:

  ```
  tsc –w
  ```

- **Detener la transpilación**:

  ```
  Ctrl + c
  ```

- **Agregar y expluir archivos a la tranpilación:**
  (agregar estas directivas al tsconfig.json)

  ```
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
  ```

Programación Estructurada

Constantes y let vs var

- **Una constante:** es un valor que no cambiara durante la ejecución del código (URL, Apikey).

```
const <nombre de contante>:tipo = valor;
```

- **Let:** Es un valor que si cambiara durante la ejecución del código

```
let <nombre de variable>:tipo = valor inicial;
```

Tipos de datos básicos

- **Booleano:** valores verdadero o falso.
  ```
  let <nombre de variable>:boolean = true o false;
  ```
- **Numerico:** valores numéricos, dec, hex, oct, bin.
  ```
  let <nombre de variable>:number = valor numérico;
  ```
- **String:** valores alfanuméricos.
  ```
  let <nombre de variable>:string = valor alfanumerico;
  ```
- **Any:** valor asignable a cualquier tipo.
  ```
  let <nombre de variable>:any = cualquier valor;
  ```

- **Arreglos:** es una colección de varios valores según su tipo.

```
let <nombre del array>: tipo de dato[] = [valores iniciales separados por comas “,”];
let <nombre del array>: Array<tipo de dato> = [valores iniciales separados por comas “,”];
```

- **Tupla:** es una colección de valores sin importar su tipo.

```
let <nombre de la tupla> = [valores iniciales separados por comas “,”];
```

- **Emun:** es una colección de valores a los que se puede acceder por indices.

```
enum <nombre del enum> { valor = indice,..,}
```

Templates literales

Los templates literales nos permiten concatenar cadenas de variables de maneras mas amigables.

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}.
I'll be ${age + 1} years old next month.`;
```

```
Resultado: Hello, my name is Bob Bobbington. I'll be
38 years old next month.
```

Funciones

- **Función sin retorno:**

```
function <nombre de función> (): void {
    Procesos…
};
```

- **Función con retorno:**

```
function <nombre de función> (): tipo de dato a retornar {
    Procesos …
    return variable de retorno
};
```

- **Función con parámetros obligatorios:**

```
function <nombre de función> (<nombre de parámetro>:tipo de dato,..): tipo
de dato a retornar {
    Procesos …
    return variable de retorno
};
```

- **Función con parámetros opcionales:**

```
function <nombre de función> (<nombre de parámetro>?:tipo de dato,..): tipo
de dato a retornar {
    Procesos …
    return variable de retorno
};
```

- **Función con parámetros por defecto:**

```
function <nombre de función> (<nombre de parámetro>:tipo de dato = <valor
por defecto>,..): tipo de dato a retornar {
    Procesos …
    return variable de retorno
};
```

- **Función asignada a una variable:**

```
let <nombre de variable> = function(): tipo de dato a retornar {
    Procesos …
    return variable de retorno
};
```

Función de fecha

- **Función fecha solo con un retorno:**

```
let <nombre de varibale> = (<nombre de parámetro>:tipo de dato): tipo de dato a retornar => variable de retorno;
```

- **Función fecha normal:**

```
let <nombre de varibale> = (<nombre de parámetro>:tipo de dato,..): tipo de dato a retornar  => {
    Procesos …
    return variable de retorno
};
```

Estructuras iterativas

- **Clasicas:** for, while.
- **For of:** opera sobre los elementos de un array u objeto.

```
let someArray = [1, "string", false];
for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}
```

- **For in:** opera sobre los índices de un array u objeto.

```
for (let i in list) {
    console.log(i); // "0", "1", "2",
}
```

- **ForEach:**

```
const someArray = [9, 2, 5];
someArray.forEach((value, index) => {
    console.log(index); // 0, 1, 2
    console.log(value); // 9, 2, 5
  }
);
```

Estructuras condicionales

- **Clasicas:** if, else, switch.
- **Operador ternario:** retorna un valor según el valor de verdad de una condición.

```
let x: number = 10, y = 20;
x > y? console.log('x is greater than y.'): console.log('x is less than or equal to y.’)
//x is less than or equal to y.
```

Métodos Console

- **.log():** Muestra cualquier tipo de mensaje en la consola.
- **.error():** Muestra mensajes de error.
- **.warn():** Muestra mensajes de advertencies.
- **.clear():** Limpia la consola.
- **.time() y .timeEnd():** Muestra el tiempo de ejecucion de una porcion de codigo.
- **.table():** Muestra un Array u objeto como una table.
- **.count():** Muestra el numero de invocaciones de una variable.
- **.group() y .groupEnd():** Agrupa impresiones en bloques.