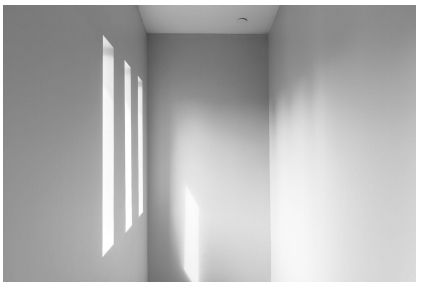
Clase I
INTRODUCCIÓN



IEEE ITBA
STUDENT BRANCH

Índice General

Índice

Qué es un lenguaje de programación

Antes de poder comenzar con la programación, tenemos que saber para qué nos sirven los lenguajes de programación.

Claramente, necesitamos poder comunicarnos con la computadora, a través de un lenguaje formal, para que esta nos ayude a resolver algún problema que nos planteemos. Para ello, están estos lenguajes de programación.

Un lenguaje de programación es un conjunto de reglas y símbolos que, juntos, permiten al programador escribir código, que luego es traducido a lo que se conoce como *código de máquina*, que es lo que la computadora puede leer y ejecutar.

Los lenguajes de programación pueden ser clasificados en diferentes tipos, basado en sus características o funcionalidades específicas, y la forma en la que están diseñados. Algunos son interpretados, otros compilados. Algunos son procedurales, otros orientados a objetos e inclusive multi paradigma.

Algunos lenguajes conocidos pueden ser Python, C++, C#, Java, JavaScript, Ruby, etc. Cada lenguaje tiene su propia sintaxis y características, y se utilizan en tipos de tareas y aplicaciones diferentes.

Ventajas y desventajas de TypeScript

Al estar basado sobre JavaScript (JS), TypeScript (TS), tiene todas las ventajas que este le otorga contra otros lenguajes. Por otro lado, incorpora características propias que lo diferencian de JS.

Ventajas

- Está fuertemente (débilmente también, como se verá más adelante) tipado. Esto implica que, cada tipo de datos, ya sean números, caracteres, strings, hexadecimales, etc., están predefinidos como parte del lenguaje de programación.
  Por ejemplo, si se crea una variable del tipo string, esta variable va a ser un string por toda la vida del programa, y tendrá permitidas ciertas operaciones, y prohibidas otras.

- Es de los lenguajes más utilizados y requeridos en la comunidad informática. Implica que existen muchas librerías, frameworks y constantes mejoras que le otorgan mayor versatilidad.
  Con este lenguaje, se crean aplicaciones tanto en móviles como en computadoras de escritorio o páginas web.

- A diferencia de JS, TS es compilado[1], lo que hace que muchos errores sean encontrados antes de la ejecución. Esto se podría tomar como una desventaja, ya que requiere tiempo de compilación, pero es relativamente menor esta contraparte. Otra característica propia de los lenguajes compilados es que son más sencillos de *debuggear* o inspeccionar para encontrar errores.

---

[1] Los lenguajes compilados son aquellos que requieren que un compilador lo "lea", para revisar que no se hayan cometido errores (de *tipado*, especialmente) antes de su ejecución.

JavaScript es un lenguaje interpretado, lo que hace que el código sea ejecutado, sin una lectura previa. Esto puede facilitar la ejecución rápida del código, ya que no requiere que un programa haya dado el visto bueno previamente. Sin embargo, se pueden encontrar errores graves durante la ejecución del mismo, cosa que no pasa con un lenguaje compilado. Esto hace que los programadores, en estos lenguajes, requieran revisar más detenidamente su código antes de correrlo. Si encuentran un error de los mencionados, pueden estar un tiempo importante revisando dónde es que ocurrió el error y por qué.

Se podría decir que TypeScript es un pseudo "compilador de compiladores", ya que del compilador traduce código de TypeScript a JavaScript cuando termina su ejecución, pero el código de JS no es compilado.

- JavaScript fue, desde un inicio, diseñado con la idea de que sea fácil de entender y utilizar, lo que lo hace un lenguaje simple. TypeScript le agrega un poco más de complejidad, al tener tipado dentro de JavaScript, pero apunta a lo mismo.
- Mejora los malos hábitos que se pueden obtener en JavaScript, al ser JS un lenguaje no tipado.
- Se pueden crear interfaces, lo que lo hace similar a otros lenguajes, y ayuda a los programadores.
- Es bueno para proyectos escalables. Esto quiere decir que posee características que lo hacen adecuado para incrementar la complejidad del código creado.
- Gran parte de lo mencionado anteriormente hace que TypeScript sea un lenguaje donde el código es más legible que en JavaScript puro.

Desventajas

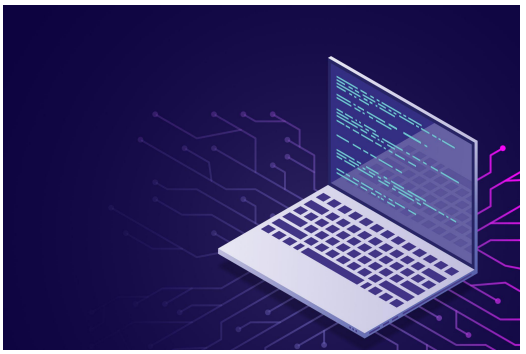
- Claramente, es más complejo de aprender que JavaScript, por toda la complejidad adicional añadida.
- Si se realizan programas que estén en el *Front End*, hay que tener en cuenta que el código TypeScript puede ser mostrado al usuario, lo que hace que la seguridad del lado del cliente sea un tema importante.
  No hay que exponer información sensible al cliente, como lo son las claves, nombres de usuarios, información de manejo de APIs, etc. Tampoco está bueno si algún código que se utilice solo para debuggear es exhibido de forma pública.
  Esto también pasa con JavaScript, por lo que comparando JS con TS, sigue ganando TS.
- Probablemente, TS no sea la mejor opción para un proyecto pequeño, por el tiempo de compilación.
- Puede resultar complejo traducir código que ya se encuentra en JS a código TS. Si ya se empieza un proyecto con JS, lo recomendable es quedarse ahí.

Como podemos ver, parece ser que tiene más ventajas que desventajas. Esto, igualmente, es relativo al conocimiento de programación, a la dedicación y a los gustos personales de cada uno. Cada lenguaje tiene lo suyo, pero con constancia y dedicación todo se puede simplificar y hacer.

Herramientas a utilizar

En un principio, vamos a necesitar de un ambiente dónde trabajar, un medio y un lugar donde podamos visualizar el código. Para ello utilizaremos las siguientes herramientas:

1. IDE (Editor de Texto)



   - WebStorm. Posee una versión gratuita para profesores, alumnos, asistencia a clases, proyectos OpenSource, cursos de entrenamiento, y un par de descuentos.
   - Visual Studio Code. Creado por Windows, gratuito y multiplataforma. El problema con el mismo es que hay que descargar extensiones al mismo para poder hacer el mismo uso de la herramienta que con WebStorm.
   - No es necesario tener las 2 herramientas al mismo tiempo. Es un tema de preferencias/comodidades/gustos y presupuesto.

2. NodeJS. Es necesario para poder crear un ambiente de servidor y correr JavaScript. Es multiplataforma, está disponible en Windows, macOS, Unix y Linux. Nos va a permitir descargar dependencias a nuestro proyecto con facilidad, como también correr nuestro código.

3. Dependencias:
   - typescript (a instalar con NodeJS).
   - ts-node (a instalar con NodeJS).
   - Más adelante, instalaremos dependencias extra a medida que vayamos avanzando con el curso.

4. Insomnia. Nos será de utilidad en la última clase, para poder probar que el programa de consultas que hagamos funcione correctamente.

En el siguiente Documento se encuentran guías de instalación de cada una de estas herramientas para iniciar el curso correctamente (hacer doble click sobre el nombre del archivo para abrirlo):

Guía de Instalación de Herramientas

Nuestro primer programa

Introducción

Para nuestro primer programa, intentaremos mantenernos con algo simple y que nos permita ir familiarizándonos con TS. Muchos deben conocer el primer paso en el aprendizaje de un lenguaje, y este es el de la creación de un programa de "Hola Mundo". Aquí aprenderemos:

- Cómo escribir la parte esencial de un programa en TypeScript.
- Cuál es el comando para imprimir información en pantalla.
- El primer tipo de datos.
- Uno de los métodos disponibles para inicializar una variable.

Primer Programa

Para realizar nuestro primer programa no será necesario crearlo de forma local, sino que podemos usar una página web que pueda compilar código TypeScript. La misma es la página oficial de TypeScript.

Para poder imprimir un mensaje, nos alcanza con utilizar la función de impresión a consola de TS, *console.log*, pasándole como argumento lo que le queramos mostrar al usuario:

```
console.log("¡Hola Mundo!");
```

Notar que, para poder pasarle como argumento una oración a mostrar, tengo que ponerla entre comillas ("). Estas pueden ser dobles tanto como simples.

También, podremos imprimir una variable de forma simple, de esta forma:

```
let mensaje: string = "¡Hola Mundo!";
console.log(mensaje);
```

Algo a resaltar es que, si bien al final de cada línea hay un ";" para indicar el fin de esa sentencia, el mismo no es necesario. El siguiente código también es válido:

```
let mensaje: string = "¡Hola Mundo!"
console.log(mensaje)
```

Pasos para creación de manera local

Pasos para la creación de este programa:

1. Crear una carpeta donde se vaya a incluir nuestro proyecto. En nuestro ejemplo, será *holamundo* (pueden usar un nombre diferente, mientras lo recuerden y tengan presente la ubicación de la carpeta).
2. Correr VSC, WebStorm, o su editor de texto de preferencia, y abrir dicha carpeta.
3. Crear un archivo de TypeScript (terminación ".ts") llamado "holaMundo.ts".
4. Escribir el código mencionado anteriormente.

La diferencia está en que el ";" se utiliza como una buena práctica de programación. En especial esto se debe a lo mencionado anteriormente, sobre que indica el fin de una sentencia. Sin los mismos, pueden haber errores relacionados a que secciones se mezclen entre sí en proyectos mucho más grandes que nuestro Hola Mundo.

Abrir una terminal (de bash, PowerShell, cmd) y ubicarse dentro de la carpeta del proyecto. También, para facilitar este paso, pueden abrir una terminal desde VSC o WebStorm, que ya estará ubicada en esta carpeta.

De no haberse descargado la dependencia ts-node, tendrán que, primero, escribrir el siguiente comando:

```
$ tsc holaMundo.ts
```

Esto compila el código y crea un nuevo archivo llamado "holaMundo.js". Este es nuestro programa pasado a JavaScript, y el cual correremos finalmente.

Una vez se haya creado el archivo "holaMundo.js", pueden correr el mismo con la siguiente sentencia:

```
$ node holaMundo.js
¡Hola Mundo!
```

De haberse descargado ts-node, estos dos últimos pasos se pueden resumir en uno solo:

Escribir lo siguiente en la terminal

```
$ ts-node holaMundo.ts
```

Notaremos que tarda un poco más la ejecución del programa que con “node holaMundo.js”. Esto se debe a que ts-node tiene que primero compilar el archivo antes de ejecutarlo.

Variables y Tipos

Ahora, vamos a desgranar un poco el código escrito con el tutorial anterior, y le iremos realizando distintas modificaciones. Recordando, habíamos indicado al programa que nuestra variable "mensaje" era del tipo "*string*" y su valor era "Hola Mundo", pero no notamos algo que tiene suma importancia. Al iniciar la variable, podemos notar la palabra "let". Esta está para mencionarle al programa que es una variable y no puede ser redefinida.

Aquí, hay mucha información que puede parecer nueva. Dentro de nuestro programa, podremos definir constantes y variables. La diferencia entre las mismas es que las constantes no pueden cambiar de valor una vez inicializadas (antes de correr el código deben estar inicializadas) y las variables pueden cambiar de valor cuantas veces quieran (restringiendo este valor a un mismo tipo de datos).

Formas de declarar: *let* vs *var*

Hay dos formas de declarar una variable, a la que llamaremos "palabra", para poder simplificar un poco las cosas:

- ***var palabra: string;***
- ***let palabra: string;***

La diferencia entre "var" y "let" es que var permite re declarar una variable, mientras que "let" no. Esto hace que "let" sea más seguro y una buena práctica frente a "var":

```
var mensaje: string = "¡Hola Mundo!"; // Declaro por
primera vez la
                                      // variable mensaje

var mensaje: string = "Otro String"; // Redeclaro
mensaje. No hay error.

mensaje = "pepito"; // Si modificamos el contenido de una
variable con
                    // el mismo tipo de datos, nunca hay
error.
```

```
let mensaje: string = "¡Hola Mundo!"; // Declaro por primera vez la
                                      // variable mensaje

let mensaje: string = "Otro String"; // Redeclaro mensaje. Ocurre un
                                     // error.

mensaje = "pepito"; // Si modificamos el contenido de una variable con
                    // el mismo tipo de datos, nunca hay error.
```

Constantes

Hay dos formas de declarar una constante:

- ***const palabra: string = "PALABRA";***
- La segunda es más para cuando tenemos un tipo complejo de datos, como los arreglos o los objetos, y es para no poder modificar los valores internos. La veremos más adelante.

```
const mensaje: string = "¡Hola mundo!";
mensaje = "pepito"; // ¡Las constantes no se pueden modificar!
```

Tipos de dato básicos

Otra de las cuestiones que ya se habrán dado cuenta es que toda variable tiene asociado un tipo de dato. Es decir, un tipo definido de información, propiedades y métodos asociados que están incorporados dentro del lenguaje. En TS, tenemos los siguientes tipos de datos básicos:

- *string* o cadena: Es utilizado para guardar todo tipo de texto. Para diferenciar un conjunto de caracteres como contenido de una variable string, se lo cierra entre comillas (").
- *number* o número: Almacena los números en el formato de punto flotante.

- *boolean* o booleano. Es una forma de almacenar valores de verdad. Puede ser únicamente *true* (verdadero) o *false* (falso).

Tipos de comparación

Operadores de Igualdad Abstracta (==)

Para comparar a una variable con algo, utilizamos el operador ==. Si bien es un operador de comparación de igualdad, se lo llama comparador de igualdad abstracto, e intentará hacer conversión de tipos para llegar al mismo tipo de datos.

Para mostrar esto, incluyendo también los valores *NaN* (no es un número) e *Infinity* (infinito), y ver un par de "incongruencias", veremos el código que está debajo. Al principio, puede ser un poco difícil de entender.

```
let a = 1, b = '01';

console.log(a == b) // true
console.log(undefined == null) // true
console.log(NaN == NaN) // false
console.log(Infinity == Infinity) // true
```

En el primer caso (a == b o 1 == '01') notaremos que el string '01' pasa a tomar formato de número. Una vez hecha esa conversión, se termina de hacer la comparación para llegar a que estos valores son iguales.

En el segundo caso, al ser *undefined* y *null* similares, como mencionamos antes, el programa nos retorna que son iguales.

Para el tercer caso, tomamos el valor que indica que un valor no es un número (utilizado con frecuencia para indicar cuando una operación matemática devuelve algún indeterminado o hubo algún error con los tipos de datos). Este caso es interesante, ya que la comparación nos devuelve falsa. Esto sigue el estándar IEEE 754, el cual indica que si el operador izquierdo o el derecho es 'NaN', entonces la comparación se vuelve falsa.

El último caso compara valores que representan al infinito positivo. En este caso, retorna verdadero, debido a que *Infinity* es un valor numérico. Entonces, al comparar dos valores numéricos iguales, el resultado es verdadero. Esto es disruptivo con el infinito matemático, ya que no tiene punto de comparación contra otro infinito matemático. En TypeScript y otros lenguajes de programación no obstante, comparar infinitos es válido.

Operadores de Igualdad Estricta (===)

Por otro lado, también existe la comparación por igualdad estricta, lo que no gastará fuerzas por convertir un tipo al otro para probar la igualdad, sino que retornará verdadero sólo si los dos elementos son absolutamente iguales:

```
let a = 1, b = '01';

console.log(a === b) // false
console.log(undefined === null) // false
console.log(NaN === NaN) // false
console.log(Infinity === Infinity) // true
```

Operadores de Desigualdad y Relacionales

Así como están los comparadores de igualdad, también están los comparadores de desigualdad, que difieren únicamente en que devuelven lo opuesto a los comparadores de igualdad. Estos operadores se representan con *!=* y *!==*. Para comparar si un número es mayor o menor que otro, se pueden utilizar < y >, como también los operadores <= y >=, dependiendo si se busca una desigualdad estricta o no.

Operadores de Negación

Un último operador es directamente la negación (*!*), que ya se mostró para indicar el "son distintos"; y la doble negación. Para explicar la negación simple, también tendremos que entender cuáles son los valores neutros de los tipos de datos. Por ejemplo, para los números, el valor neutro es el 0, y para los strings, el valor neutro es el string vacío (''). *null* y *undefined* cuentan como valores neutros también.

Si tenemos una variable no neutra y le aplicamos la negación, nos devolverá falso, y si tenemos una variable neutra, la negación nos devolverá verdadero. La doble negación sirve para saber el valor de verdad de un valor en sí (o sea devolver si este valor no es neutro, no es vacío y está definido).
A continuación les dejamos ejemplos sobre el resultado de aplicar la negación a distintos tipos de variables:

```
console.log(0, !0, !!0); // 0, true, false
console.log(1, !1, !!1); // 1, false, true
console.log(null, !null, !!null); // null, true, false
console.log(undefined, !undefined !!undefined); // undefined, true,
false
console.log("", !"", !!""); // "", true, false ("" es la representación
del string vacío)
```

```
console.log("hola", !"hola", !!"hola") // hola, false, true
```

Para un mayor entendimiento sobre los tipos de comparación, pueden leer la bibliografía adicional.

Ciclos y condicionales

Algo que aparece mucho en la programación es la iteración. Normalmente nos encontramos en la situación de querer hacer algo repetidas veces, cambiando poco y nada un mismo paso. Una iteración simple sería, por ejemplo, cuando uno quiere mostrar en pantalla los números del 1 al 10. Para esto uno podría hacer algo semejante a:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
console.log(6);
console.log(7);
console.log(8);
console.log(9);
console.log(10);
```

El problema de esto es que estamos repitiendo mucho código y, si el día de mañana nos piden imprimir en pantalla los números del 1 al 100.000, tendríamos que pasarnos una tarde entera, por lo menos, escribiendo cada sentencia. La solución es usar un “loop” (bucle).

Queremos llegar a algo del siguiente estilo:

Paso	Acción
1	Iniciar las variables
2	Verificar la condición
3	Si la condición es verdadera, ejecutar el cuerpo

4	Actualizar la variable
5	Volver al paso 2
6	Si la condición es falsa, terminar

Bucle *do while*

Hay múltiples formas de realizar un *loop* o bucle. Por lo general, un loop repite una porción de código hasta que se alcanza cierta condición (por ejemplo, hasta que el número a mostrar en pantalla sea el 100.000). El primero que veremos se llama *do {...} while:*

```
do{
    expresión;
}while(condición);
```

Una forma (incorrecta) de implementar lo que queremos hacer, usando este bucle, se muestra abajo:

```
let i = 1;
do{
    console.log(i);
}while(i++ <= 10);
```

No queríamos que se corra el do {...} while para el número 11, pero pasó de todas formas, porque recién al finalizar cada iteración chequea la condición de cierre, mientras que el for y while normales lo hacen al inicio de cada iteración.

Para ello, incrementaremos el valor de i antes de entrar al condicional:

```
let i = 1;
do{
    console.log(i);
    i++;
}while(i <= 10);
```

Bucle *while*

También hay *loops* del tipo *while*. Son similares, pero se utilizan en distintas ocasiones:

```
while(condición){
      expresión;
}
```

Para entrar a la condición del while, uno tendría que ya tener la variable que se utilice inicializada.
Una forma de hacer el programa anterior (de forma incorrecta) con un *while* es la siguiente:

```
let i = 1;
while(i <= 10){
    console.log(i);
}
```

Nos hemos olvidado de modificar el valor de *i*, haciendo que el programa corra eternamente. Hay que poner esta instrucción dentro del bloque de código contenido en el bucle.

```
let i = 1;
while(i <= 10){
    console.log(i);
    i = i + 1;
    // o i += 1;
    // o i++;
}
```

Notar que, para incrementar la variable, se puede usar i = i + 1, como i += 1 como i++. i++ se puede utilizar para incrementar la variable de a uno. No existe para

incrementarla de a dos o etc.
También se puede decrementar de a uno con i-- o i -= 1

Bucle *for*

Por último, el último *loop* que veremos hoy se llama *for* y sigue el siguiente formato:

```
for([expresiónInicial]; [expresiónCondicional]; [expresiónDeActualización]){
      instrucción;
}
```

En nuestro código, habría que poner lo siguiente:

```
for(let i = 1; i <= 10; i++){
      console.log(i);
}
```

De esta forma le estamos indicando lo siguiente al programa:

1) Inicia la variable *i* en 1.
2) En cada iteración, chequear que se cumpla la condición de que *i* sea menor o igual que 10.
3) En caso de que sea así, imprimir *i*.
4) Sino, termina el *for*.
5) Al terminar cada iteración, *i* incrementará su valor en 1.

Entonces, *i* tomará valores entre 1 y 11. Entre 1 y 10 imprime el número en la consola, mientras que para el valor 11, la condición de que sea menor o igual a 10 no se cumple más, por lo que sale del *for*, sin imprimirlo.

Preferentemente, el *for* es el *loop* de preferencia mientras programamos en TypeScript. De todas formas, puede que hayemos que este excede lo que necesitamos hacer, por lo que podríamos optar por el *while*. El *do {...} while* no se usa tan seguido en TypeScript.

Claramente, ustedes pueden optar por el que más les guste sin problemas, mientras tengan en cuenta las advertencias que mencionamos en cada explicación.

Condicionales

If-else

Para poder correr código dependiendo del resultado de una condición, deberemos hacer uso de la sentencia "***if(condicion) { operación } [else [if ...] ...]***".

La estructura de los condicionales sigue los pasos de esta tabla:

Paso	Acción
1	Verificar la condición
2	Si la condición es verdadera, ejecutar la declaración "if"
3	Si la condición es falsa, ejecutar la declaración "else" (si existe)
4	Terminar la declaración condicional

Para probar esto, un ejemplo simple podría ser crear una variable *numero*, asignarle un número dentro del programa, y chequear si ese número es par. En el caso de que sea par, tendría que imprimir el número.

Utilizaremos la operación *módulo* (a % b, siendo a y b números enteros, y b distinto de 0), que devuelve el resto de dividir a por b, para evaluar la condición anterior. Para indicar que sea par, tenemos que pedir que el resto de la división entera por 2 sea 0. Por lo tanto, usaremos el operador ===. Internamente, nuestro nuevo bucle nos quedaría de la siguiente manera:

```
const numero1 = 5;
if(numero1 % 2 === 0) {
    console.log(numero1);
}
```

Como podemos notar, este ejemplo no imprimirá nada, debido a que 5 es impar. Si lo cambiamos para que imprima solo los impares, en ese caso sí imprimiría algo:

```
const numero2 = 7;
if(numero2 % 2 === 1) {
    console.log(numero2);
}
```

Para un ejemplo un poco más complejo, si queremos que entre el 1 y el 10 se impriman únicamente los números pares, queremos que 'i' se pueda representar de la forma *i = 2 * k*, siendo *k* un número entero. *i tomará los valores* 2, 4, 6, 8, 10.

```
for(let i = 1; i <= 10; i++){
    if(i % 2 === 0){
        console.log(i);
    }
}
```

Para conectar condiciones, podemos usar el "y" (*and*) y el "o" (*or*) lógicos. El formato dentro de TypeScript para estos es *&&* y ||, respectivamente. Para conectar dos condiciones if, y correr una condición si no se cumple la condición anterior, deberemos concatenar cada if con una secuencia "if {...} else if {...} else if {...} else". Por ejemplo, si queremos que corte el programa o "salte un error" cuando *i* sea un par mayor a 6, podríamos hacer la siguiente operación:

```
for(let i = 1; i <= 10; i++){
    if(i <= 6){
        console.log('El número no es mayor que 6');
    }else if(i % 2 === 0){
        console.log(i);
    }
}
```

Para negar una condición, deberemos utilizar el operador ! (not). Esto hará que, si una condición nos devuelve *verdadero*, ! condición otorga *falso*.

Si bien se puede construir toda la lógica de un programa con bucles y condicionales tipo if-else, existen otras herramientas dentro de TS que simplifican tomas de decisiones un poco más intrincadas.
Supongamos que queremos implementar una función que imprima qué día de la semana es.

Con lo que vimos hasta ahora, por cada tipo distinto de día de la semana, habría que escribir un *if* en el programa, lo que no quedaría muy lindo. No obstante, si nosotros optamos por una seguidilla de *if else*, nos quedamos con el siguiente código:

```
for(let diaDeSemana = 1; diaDeSemana <= 7; diaDeSemana++) {
    if(diaDeSemana === 1) {
        console.log("Hoy es Lunes");
    } else if(diaDeSemana === 2) {
        console.log("Hoy es Martes");
    } else if(diaDeSemana === 3) {
        console.log("Hoy es Miércoles");
    } else if(diaDeSemana === 4) {
        console.log("Hoy es Jueves");
    } else if(diaDeSemana === 5) {
        console.log("Hoy es Viernes");
    } else if(diaDeSemana === 6) {
        console.log("Hoy es Sábado");
    } else { // diaDeSemana === 7
        console.log("Hoy es Domingo");
    }
}
```

Switch-case

Como podemos ver, estamos repitiendo mucho código, lo cual no es lo ideal. Una buena práctica en estos casos es la utilización del '*switch case*'. La idea del 'switch' es que en base a un valor se elija una entre varias opciones. La diferencia con el 'if' es que la sintaxis es más limpia y clara para el que lea nuestro código:

```
switch(valor){
      case 1:
            // código a ejecutar si se cumple el caso 1.
            break;
      case 2:
            // código a ejecutar si se cumple el caso 2.
            break;
      ...
      default:
            // código a ejecutar si no se llega a algún caso de interés.
            break;
}
```

Para entender el uso del *switch-case*, podríamos verlo, en un inicio, con un ejemplo muy simple, que imprime qué día de la semana es:

```
const diaDeSemana: number = 3;
switch(diaDeSemana){
  case 1:
      console.log("Hoy es Lunes.");
      break;
  case 2:
      console.log("Hoy es Martes.");
      break;
  case 3:
      console.log("Hoy es Miércoles.");
      break;
  case 4:
      console.log("Hoy es Jueves.");
      break;
  case 5:
      console.log("Hoy es Viernes.");
      break;
  case 6:
      console.log("Hoy es Sábado.");
      break;
  case 7:
```

```
      console.log("Hoy es Domingo.");
      break;
}
```

En este caso, nos imprimiría “Hoy es Miércoles”, debido a que diaDeSemana es 3.

Al aplicar un *switch-case* en nuestro problema, obtenemos el siguiente código:

```
for(let diaDeSemana = 1; diaDeSemana <= 7; diaDeSemana++){
    switch(diaDeSemana){
      case 1:
            console.log("Hoy es Lunes.");
            break;
      case 2:
            console.log("Hoy es Martes.");
            break;
      case 3:
            console.log("Hoy es Miércoles.");
            break;
      case 4:
            console.log("Hoy es Jueves.");
            break;
      case 5:
            console.log("Hoy es Viernes.");
            break;
      case 6:
            console.log("Hoy es Sábado.");
            break;
      case 7:
            console.log("Hoy es Domingo.");
            break;
    }
}
```

Como puede verse, es una forma mucho más clara que concatenar ‘if else’ para el caso de múltiples chequeos de formato similar a la misma información. Una buena práctica es siempre indicar un valor de default, en especial para valores no esperados o que no nos pueden llegar a interesar. En nuestro ejercicio, como solo hay 7 valores posibles, hay un caso para cada uno, y no hay valores que puedan romper nuestro código, no tiene sentido agregar el default.