Universidad Nacional Experimental Simón Bolívar
Lenguajes de Programación I
Prof. Ricardo Monascal

Caracas, 11 de Marzo 2021

Alumno y Carné:
Victoria Torres 12-11468

Examen 2: Pregunta 1

1)

(a) <u>De una breve descripción del lenguaje escogido</u>

TypeScript, es un lenguaje de programación libre y de código abierto desarrollado, y mantenido, por Microsoft. Es un superconjunto de JavaScript, por lo cual puede ejecutar programas de este lenguaje también, es decir que los programas de JavaScript son programas válidos de TypeScript por lo cual podemos encontrar proyectos existentes en JS con código de TS integrado y viceversa. Uno de los beneficios de esta característica del lenguaje es que pone a disposición el enorme ecosistema de librerías y frameworks que existen para JS.

El lenguaje como tal sigue el patrón de JavaScript utilizando la misma sintaxis y semántica y también es un lenguaje multiparadigma, sin embargo, agrega funcionalidad y sintaxis orientada a objetos, clases y escritura vistos en lenguajes como C# y Java. Esencialmente añade a JS tipos estáticos y objetos basados en clases

El código TS se guarda en archivos con extensión ".ts" y su compilador esta escrito también en TS, compilado a JavaScript.

i. Diga qué tipos de datos posee y qué mecanismos ofrece para la creación de nuevos tipos (incluyendo tipos polimórficos de haberlos).

Como antes vimos que TypeScript es un superconjunto de JavaScript entonces no es de extrañarnos que comparta tipos de datos que este posee. En TS los tipos primitivos son "number", "string", "boolean", "symbol" , "null" y "undefined", es decir que son los únicos tipos que no son específicamente objetos (aunque como JS envuelve los 3 primeros con sus correspondientes objetos String, Number y Boolean entonces tenemos la opción de declarar variables de estos tipos usando por ejemplo: let x = new String("abc") y sería válido para TS). A parte de éstos, tenemos otros tipos y también el tipo Object que nos dará la posibilidad de crear nuevos tipos cuando buscamos construir elementos de este tipo. A continuación, veremos un conjunto de los más conocidos:

- **El tipo Null:**

  Tal y como en JS, este tipo de datos solo puede tener un valor válido: "null", por lo cual no podrá contener otros tipos de datos como un número o un string. Por otra parte, si establecemos una variable a null borraremos su contenido (si tuviese alguno).

  Cuando en un proyecto vamos a trabajar con TS tendremos un archivo llamado "tsconfig.json" y es el que indicará que se está trabajando con este lenguaje. Suele colocarse en la raíz de carpetas del proyecto y en se situa un **JSON** con todas las configuraciones de trabajo para el transpilador de TS.

  TS ofrece la posibilidad de configurar en "tsconfig.json" el indicador "strictNullChecks". Si se coloca como "true" entonces el valor "null" solo podrá asignarse a las variables con tipo null. Normalmente suele estar desactivado por defecto (es decir que esta como "false") por lo cual se puede asignar el valor null a variables con otros tipos como "number" o "void".

  Por ejemplo:

  Si colocamos strictNullChecks como true, entonces al hacer:

  let a: null = null; // No habrá problemas porque "a" es de tipo null.

  let b: undefined = null; // Nos da error porque "b" es de tipo undefined y no null.
  let c: number = null; // Nos dará Error
  let d: void = null; // Nos dará Error.

  Si dejamos strictNullChecks como false, entonces al hacer:

  let a: null = null; // Éxito!
  let b: undefined = null; // Éxito!
  let c: number = null; // Éxito!
  let d: void = null; // I-can-talk-wachington-tu! (xD)

- **El tipo Undefined (Indefinido):**

  Cualquier variable a la que no se le haya especificado su valor se establece como "undefined". Sin embargo, también podemos establecer explícitamente el tipo de una variable como "undefined" como se vio en el ejemplo anterior en el tipo null donde apareció:

  let b: undefined = null;

  Hay que tener en cuenta que una variable cuyo tipo este configurado como "undefined" solo podrá tener "undefined" como su valor.

Por otro lado, también tenemos que cuando el indicador “strictNullChecks” está configurado como false en “tsconfig.json” entonces podríamos asignarle a variables de otros tipos “undefined” ya que cuando este indicador es “false” tanto “null” como “undefined” son efectivamente ignorados y podemos asignarlos a variables con otros tipos sin tanto problema, sin embargo, esto podría decantar en errores inesperados en tiempo de ejecución si no tenemos cuidado. Pero cuando tenemos “strictNullChecks” como “true”, tanto “null” como “undefined” tienes sus tipos distintivos por lo cual tendremos un error de tipo si tratamos de asignárselos a variables en las que un tipo de valor en concreto se espera (aunque para el caso de “undefined” hay un tipo de variable a la que se lo podremos asignar sin problema aún con el indicador activo y sabremos qué tipo en el siguiente).

Por ejemplo:

Si colocamos strictNullChecks como true, entonces al hacer:

```
let a: undefined = undefined;    // No hay problema porque se corresponden los tipos.
let b: undefined = null;         // Error porque como se vimos antes, no se corresponde null con undefined.
let c: number = undefined;       // Error porque en c se espera un número y undefined no es de ese tipo.
let d: void = undefined;         // Éxito, por razones que veremos en el tipo Void (No Spoilers! xd).
```

Si dejamos strictNullChecks como false, entonces al hacer:

```
let a: undefined = undefined;   // No hay problema, porque se corresponde con el tipo de la variable.
let b: undefined = null;        // Tampoco hay problema porque tenemos libertar de asignar null o otras.
let c: number = undefined;      // Tampoco hay problem, porque podemos asignar undefined con libertad.
let d: void = undefined;        // Tampoco tiene problema, en el próximo tipo veremos por qué.
```

- **El tipo Void:**

  Se usa para indicar la falta de un tipo para una variable. Establecer variables para que tengas un tipo “void” puede que no sea muy útil pero podemos usarlo para establecer el tipo de retorno de funciones que no devuelven nada y cuando se usa con variables, este tipo solo puede tener dos valores válidos: “null” (si el indicador “strictNullChecks” no se encuentra activado) o “undefined” (que no se ve afectado por el indicador “strictNullChecks”).

Ejemplos:

Si colocamos strictNullChecks como true, entonces al hacer:

```
let a: void = undefined;  // Éxito porque no se afecta la posibilidad de asignar undefined
let b: void = null;          // Error, directo porque esta activado el indicador.
let c: void = 3;             // Error, porque a void solo se le asigna null o undefined.
let d: void = "apple";   // Error (copy-paste del error anterior)
```

Si dejamos strictNullChecks como false, entonces al hacer:

```
let a: void = undefined; // Éxito con el indicador activado o no
let b: void = null;          // Éxito para null estando el indicador desactivado.
let c: void = 3;             // Error forever porque “qué pretende ese 3?”
let d: void = "apple";   // Error porque void no admite nada que no sea null o undefined
```

Lo podemos usar entonces para funciones que no queremos que regresen nada:

```
function holaMundo(): void{
        console.log(‘Hola Mundo!’)
}
```

Y también si quisiéramos retornar null o undefined:

```
 function lafuncion(a: number, b:number) : void{
        return undefined
}
```

- **El tipo Booleano:**
  Solo tiene dos valores válidos, “true” o “false”. Podemos tener entonces, por ejemplo:

```
let a: boolean = true;
let b: boolean = false;
let c: boolean = 23;       // Error :v
let d: boolean = "blue"; // Error :v
```

- **El tipo Número:**

  Usado para representar tanto enteros como valores de punto flotante. Sin embargo, es importante recordar que todos los números están representados internamente como valores de coma flotante y también podemos especificarlos como literales, hexadecimales, octales o binarios (Estos dos últimos introducidos en ES6 que es el estándar más reciente de JS por lo cual podría dar como resultado una salida de código en JS diferente según la versión a la que se esté apuntando cuando usemos TS).

  Adicionalmente tenemos 3 valores simbólicos especiales adicionales que se encuentran bajo el tipo de number: “+Infinity”, “-Infinity” y “NaN”.

  Como ejemplo de este tipo tenemos:

  Con “strictNullChecks” como true tenemos:

```
let a: number = undefined;     // Error por el indicador activo
let b: number = null;          // Error por el indicador activo
let c: number = 3;             // Asignación normal de un entero
let d: number = 0b111001;  // Con Binario
let e: number = 0o436;       // Con Octal
let f: number = 0xadf0d;     // Con Hexadecimal
let g: number = "cat";       // Error si intentamos cosas raras
```

  Con “strictNullChecks” como false tenemos:

```
let a: number = undefined;  // Éxito por indicador desactivado
let b: number = null;          // Éxito por indicador desactivado
let c: number = 3;             // Asignación normal de un entero
let d: number = 0b111001; // Con Binario
let e: number = 0o436;       //Con Octal
let f: number = 0xadf0d;    //Con Hexadecimal
let g: number = "cat";       // Error si seguimos intentando cosas raras
```

  Como algo interesante a tomar en cuenta, si la versión destino de JS se establece en ES6, el código anterior se compilará a el siguiente JS:

```
let a = undefined;
let b = null;
let c = 3;
let d = 0b111001;
let e = 0o436;
let f = 0xadf0d;
let g = "cat";
```

Hay que considerar que las variables de JS todavía se declaran con let (introducido en ES6) y nos fijamos que no habrá mensajes de error relacionados con los tipos de variables diferentes porque el código de JS no tiene conocimiento de los tipos que usamos en el código TS.

Si la versión destino de JS estuviese, por ejemplo, configurada como ES5 el mismo código de TS se compilaría como sigue:

```
var a = undefined;
var b = null;
var c = 3;
var d = 57;
var e = 286;
var f = 0xadf0d;
var g = "cat";
```

Y vemos como en este caso todas las ocurrencias de la palabra clave “let” fueron cambiadas a “var” y tanto el número Binario como el Octal fueron cambiados a sus formas decimales.

- **El tipo String:**

  Lo usamos para guardar información “textual”. TS, al igual que JS utiliza comillas dobles (“), así como comillas simples (‘) para expresar esa información como una cadena, además éstas pueden contener cero o más caracteres entre comillas. Por ejemplo:

  Con “strictNullChecks” como true tenemos:

```
let a: string = undefined;  // Error
let b: string = null;       // Error
let c: string = "";         // Éxito con el c onocido String vacío
let d: string = "y";        // Éxito
let e: string = "building"; // Éxito
let f: string = 3;          // Error porque intentamos asignarle un number
let g: string = "3";        //Éxito
```

  Con “strictNullChecks” como false tenemos:

```
let a: string = undefined; // Éxito por razones conocidas
let b: string = null;      // Éxito
let c: string = "";        // Éxito
let d: string = "y";       // Éxito
let e: string = "building";// Éxito
let f: string = 3;         // Error por mala asignación.
```

```
let g: string = "3";      // Éxito
```

TS también admite “Plantillas de String” o “Plantillas de Literales”. Éstas permiten incrustar expresiones en un String. Están encerradas por el carácter “retroceso” (`) en lugar de comillas dobles o simples. Éstas también se presentaron en ES6, lo cual significa que así como vimos con los “number” vamos a obtener diferentes JS en función de la versión que estemos usando. Entonces si por ejemplo en TS tenemos:

```
let e: string = "building";
let f: number = 300;

let sentence: string = `The ${e} in front of my office is ${f} feet tall.`;
```

Y al compilar obtendremos

En JS configurada con ES5:

```
var e = "building";
var f = 300;
var sentence = "The " + e + " in front of my office is " + f + " feet tall.";
```

En JS configurada con ES6:

```
let e = "building";
let f = 300;
let sentence = `The ${e} in front of my office is ${f} feet tall.`;
```

Vemos como la plantilla del literal cambió a un String regular en ES5 y se adaptó a ES6 sin problema también por lo cual confirmamos como TS hace posible que se usen las funciones más recientes de JS sin preocuparse por la compatibilidad.

- **Los tipos Array y Tuple:**

  Se pueden definir arreglos de dos formas diferentes. El primer método especificando el tipo de elementos de array seguido por “[]” para denotar el arreglo de ese tipo. El segundo método es utilizar el tipo array genérico “Array<elemType>”.

  Algo con lo que se tiene que tener cuidado es que si especificamos “null” o “undefined” como uno de los elementos de un arreglo, esto puede producir errores cuando el indicador “strictNullChecks” este activado.

  A continuación, veremos ejemplos:

  Con “strictNullChecks” como false tenemos:

Con el método 1:

```
let a: number[] = [1, 12, 93, 5]; //Un arreglo de numeros
let b: string[] = ["a", "apricot", "mango"]; //Un arreglo de strings
let c: number[] = [1, "apple", "potato"]; // Error porque apple y potato no son números.
```

Con el método 2:

```
let d: Array<number> = [null, undefined, 10, 15];
let e: Array<string> = ["pie", null, ""];
```

Observamos que podemos asignar “null” y “undefined” a posiciones en el arreglo, sin importar su tipo, y lo permite porque el indicador esta desactivado.

Con “strictNullChecks” como true tenemos:

```
let a: number[] = [1, 12, 93, 5]; //Arreglo de números normal
let b: string[] = ["a", "apricot", "mango"]; //Arreglo de String
let c: number[] = [1, "apple", "potato"]; // Error anteriormente visto

let d: Array<number> = [null, undefined, 10, 15]; // Error por el indicador
let e: Array<string> = ["pie", null, ""]; // Error por el indicador
```

Vemos que aquí estos últimos casos si nos resultan en error por las razones explicadas para el tipo null y el tipo undefined anteriormente y el comportamiento que tienen cuando el indicador este activo.

El tipo Tuple, permite crear un “arreglo” donde hay un número fijo de elementos que conocemos de antemano. El tipo de los elementos en cuestión será uno de los tipos que hayan sido especificados cuando se construyó la tupla.
Por ejemplo:

```
let a: [number, string] = [11, "monday"]; //Una tuple (numero,string)

let b: [number, string] = ["monday", 11]; // Error

let c: [number, string] = ["a", "monkey"]; // Error
let d: [number, string] = [105, "owl", 129, 45, "cat"]; //Éxito
let e: [number, string] = [13, "bat", "spiderman", 2];  //Éxito

e[13] = "elephant"; // Éxito al asignar
e[15] = false;      // Error
```

Para todas las tuplas establecimos que el tipo de su primer elemento fuese “number” y el segundo “string”, por lo cual como solo hemos especificado estos tipos para los dos primeros elementos, el resto de ellos podrá ser, de igual forma, o “String” o “Number” en cualquier orden (como vemos por ejemplo en “d” y “e”).

Por esto mismo vemos que “b” y “c” dan error, ya que estamos tratando de colocar un string para el primer elemento de la tupla y el tipo pasa ese primer elemento debe ser un numero.

También podemos ver que en “e[15]” la asignación falla porque estamos intentando que un elemento de la tupla sea un booleano, cuando especificamos que esta sólo contendría String o Numbers.

- **El tipo Enum:**

  Este tipo de dato no esta originalmente en JS sino que TS lo introduce. Éste tipo permite crear una colección de valores relacionados con nombres.
  Por ejemplo:

```
enum Animals {cat, lion, dog, cow, monkey}
let c: Animals = Animals.cat;

console.log(Animals[3]);     // nos da “cow”, que esta en la posción 3
console.log(Animals.monkey); // nos da “4 ”, la posición de Monkey
```

  Por defecto la numeración de un “enum” comienza en 0, pero si se quisiera podría establecerse un valor diferente para el primero u otros miembros de forma manual. Esto haría que cambie el valor de todos los miembros que lo siguen aumentando sus valores en 1. Por otra parte, también se podrían establecer todos los valores de un enum en forma manual.cAlgunos ejemplos serían:

```
enum Animals {cat = 1, lion, dog = 11, cow, monkey}
let c: Animals = Animals.cat;

console.log(Animals[3]); // undefined
console.log(Animals.monkey); // 13
```

  En este caso podemos observar que se le colocó manualmente su posición a cat y a dog. Si quiero ver el valor que tengo en la posición 3 obtengo “undefined” porque si bien al asignarle a cat la posición 1, dog tendría que haber sido asignado a 3, como luego manualmente colocamos dog en la posición 11, entonces en la posición 3 no hay nada definido realmente y por ende recibo esto como resultado.

Por otra parte si esta vez vemos en qué posición esta Monkey obtenemos 13 porque como dog ahora está en la posición 11, el resto de elementos en el “enum” ajustan sus posiciones según la anterior (el descontrol :V).

- **Los tipos Any y Never**

  Cuando escribimos un programa donde el valor de una variable es determinada por los usuarios o el código escrito en una biblioteca de terceros, no podremos establecer el tipo de dicha variable correctamente ya que la variable podría ser de cualquier tipo (String, Booleano, Number, etc). Por este problema en particular nació el tipo Any, que puede aceptar todos los tipos de valores.
  Por ejemplo:

```
let a: any = "apple";
let b: any = 14;
let c: any = false;
let d: any[] = ["door", "kitchen", 13, false, null];

b = "people";
```

  Podemos ver aquí que a una variable del tipo any podemos asignarle un valor de cualquier otro tipo. Hasta podemos ver como no dio error que a la variable “b”, que inicialmente guardaba un number, le asignáramos luego un string.

  Este tipo también suele ser muy útil cuando estamos creando arreglos con elementos de tipos mixtos.

  Luego, el tipo Never se utiliza para representar valores que nunca se supone que ocurran. Por ejemplo, podemos asignar never como el tipo de retorno de una función que nunca devuelve algo, esto puede suceder por ejemplo cuando una función siempre arroja un error o cuando esta atrapada en un ciclo infinito.
  Por ejemplo:

```
let a: never; // Éxito
let b: never = false; // Error
let c: never = null; // Error
let d: never = "monday"; // Error
```

  Vemos que en variables, si son never nos darán error al tratar de asignarles algún valor de otro tipo de dato porque son cosas que pueden ocurrir normalmente. Por lo cual este uso no tiene mucho sentido para este tipo de dato. Pero cuando vemos:

```
function stuck(): never {
  while (true) {
  }
}
```

```
function errorFn(message: string): never {
  throw new Error(message);
}
errorFn('Error occurred');
```

Podemos encontrarle utilidad en funciones que no terminan como las que vemos arriba.

- **El tipo Símbolo (Symbol)**

  Su característica principal es que pueden actuar como identificadores, que por su naturaleza deben ser únicos e inmutables. Aunque es un tipo de dato primitivo, también tiene una envoltura en el objeto "Symbol". Pero, un symbol no puede generarse con una notación literal como un string.
  Por ejemplo:

```
let x = Symbol();
console.log(typeof x); // "symbol"
```

  Debemos utilizar esta notación y es importante ver que no se usa la palabra "new" porque Symbol() no es un constructor. De hecho si hiciéramos:

```
let y = new Symbol(); //Obtenemos: sTypeError: Symbol is not a constructor
```

  Lo más importante de este tipo es que debemos entenderlo como un "identificador" y sabemos que para estos no deberían existir dos identificadores iguales, por lo tanto en este caso para los símbolos es igual, la idea es que no existan dos símbolos iguales por lo cual éstos tienen valores únicos y no es posible modificar su contenido (son inmutables).

- **El tipo Union:**

  Este nos permite trabajar con varios tipos de datos que nosotros especifiquemos sobre una variable de la siguiente forma:

```
let numero: string | number | Boolean = 5;
numero = "tres";
numero = false;
```

  Observaremos que temenos éxito en las 2 asignaciones porque el tipo unión hace que especifiquemos que tipos puede ser la variable en cuestión. Por lo que esta puede recibir number, string y boolean. Si intentamos pasarle algo diferente de lo definido, entonces nos arrojará error.

- **El tipo Objeto (Object):**

  Este tipo es el que representa tipos no primitivos, es decir, cualquier cosa que no sea “number”, “string”, “boolean”, “symbol” , “null” o “undefined”.

  En TS, Object es quien nos permite definir tipos de datos personalizados con la composición de nuestra elección. Hay dos formas en las que podemos crear Object en TS. La primera es usar el nuevo constructor “Object()” y la segunda es utilizar corchetes “{}”
  Por ejemplo:

```
let planet = new Object();
let planet = {};
```

  Sin embargo, un objeto puede tener propiedades definidas dentro de ellos, de manera que podemos definir tipos mas complejos y para esto hacemos uso de “interfaces” y de “clases” al momento de crear nuestros objetos.

  Por ejemplo:

  Si utilizamos una interfaz entonces quedaría de la siguiente forma:

```
interface Cliente {
 nombre: String;
 cif: String;
 direccion: String;
 edad: number;
}
```

  De esta manera, una vez definida la interfaz podemos crear los objetos de dicha interfaz. Por ejemplo, podríamos decir:

```
let cliente: Cliente = {
 nombre: “Anita”;
 cif: “elcif”;
 direccion: “su dirección”;
 edad: 26;
}
```

  Sin embargo, también podemos simplemente crear una variable asignando la interfaz como si fuera un tipo:

```
let cliente: Cliente;
```

  A partir de aquí el editor nos avisaría cuando un valor que se quiera asignar no cumpla con lo establecido en la interfaz, por ejemplo, si intentamos hacer:

Cliente = {name:'test'}

Esto nos arrojara un error, puesto que el objeto Cliente no posee una propiedad "name" (en todo caso es "nombre").

Si utilizamos una clase para definir el tipo de los objetos que vamos a crear todo puede parecer más familiar, sobre todo si se está acostumbrado a programación orientada a objetos. Esto quedaría de la siguiente forma:

```
class Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
```

Y tal y como vimos con las interfaces, tenemos la opción de crear un objeto del tipo definido por la clase y poblar sus propiedades o simplemente declarar un cliente usando el tipo de la clase cliente de la forma:

```
let cliente: Cliente;
```

El decidir cuál de estas dos vías utilizaremos al momento de construir los tipos queda a las necesidades del implementador y del programa que se esta construyendo por ejemplo es muy común usar interfaces, desprovistas de inicialización, en cambio las clases si tienen un constructor formal y para algunos casos será más conveniente tenerlo.

Por otra parte, una razón importante por la que se utilizan las interfaces es para conseguir realizar polimorfismo, veamos con un ejemplo como funciona esto acá:

Partimos teniendo en mente que estaremos trabajando para un módulo de conección, por lo cual plantearemos una interfaz y algunas clases.

Primero tenemos al interfaz Connector:

```
interface Connector{
   doConnect(): boolean;
}
```

Ahora, implementamos de Connector una clase para comunicación por Wifi. Entonces tendremos una clase concreta WifiConnector, con su propia implementación y que será del tipo Connector:

```
export class WifiConnector implements Connector{

  public doConnect(): boolean{
    console.log("Connecting via wifi");
    console.log("Get password");
    console.log("Lease an IP for 24 hours");
    console.log("Connected");
    return true
  }

}
```

Finalmente creamos nuestra clase “System” que tiene un componente “Conector” con su propia implementación (a esto se le llama “inyección de dependecia” que es un patrón de diseño)

```
export class System {
  constructor(private connector: Connector){ #inject Connector type
    connector.doConnect()
  }
}
```

Cuando colocamos “constructor(private connector: Connector)” resulta importante aquí. Connector es una interfaz y como tal debe tener “doConnect()” por como la definimos, y por ser interfaz la clase System tiene mucha más flexibilidad. Podemos pasar cualquier tipo que tenga implementada la interfaz Connector y esto ayudará a futuro ya que si por ejemplo en la actualidad queremos agregar al modulo “Conección Bluetooth” podríamos crear la clase:

```
export class BluetoothConnector implements Connector{

  public doConnect(): boolean{
    console.log("Connecting via Bluetooth");
    console.log("Pair with PIN");
    console.log("Connected");
    return true
  }

}
```

Entonces acá podemos ver que hicimos implementaciones para Wifi y Bluetooth con clases separadas, cada una con su propia manera de “conectarse”. Sin embargo, como ambas han implementado el tipo Connector, entonces son del tipo

Connector. De esta forma, podríamos pasarle a cualquiera de ellas a System como parámetro para el constructor.

Esto es a lo que llamamos polimorfismo, la clase System como tal no sabe si será Bluetooth o Wifi, incluso hasta podríamos agregar otras clases que implementen a la interfaz Connector y también podríamos pasárselas a System.

Esto se llama “duckTyping”, el tipo Connector ahora es dinámico ya que “doConnect()” es solo un “placeholder” y el desarrollador puede implementarlo como propio.

De esta manera si tenemos en System “constructor(private connector: WifiConnector)” entonces la clase System se acoplará de forma estrecha sólo con WifiConnector y asó la interfaz nos da opción de hacer esto por polimorfismo.

Las interfaces sob un buen aliado cuando queremos crear nuevos tipos, podemos ampliarlas lo que hace que las adaptemos a medida que lo necesitemos.

Sin embargo, también se puede mencionar una forma de crear nuevos tipos en TS diferente a las dos anteriores y es con el uso de **“type”.** Bueno, tal vez “crear un nuevo tipo” no es la mejor forma de describirlo, mas adelante veremos que lo que hacemos realmente es crear un nuevo “alias de tipo”.

A los type no le podemos extender ni aumentar sus capacidades como por ejemplo podemos hacer usando las interfaces pero es una forma de añadir tipos personalizados, que habríamos creado previamente y que asignaríamos a nuestro nuevo type. Con type podemos ser más estrictos. Si por ejemplo teníamos una interfaz:

```
interface Cat {
 name: string;
 legs: number;
 isDogFriendly: boolean;
}
```

Si usamos type, escribimos como tal:

```
type Cat = {
   name: string;
}

type DogFriendly = {
   isDogFriendly: boolean;
}
```

```
type CatAndDogFriendly = Cat & DogFriendly;
```

Es importante entender que en TS tenemos alias de tipo o (type aliases), con los cuales podemos crear nuevos nombres para un tipo pero no definimos un nuevo tipo per se.

Cuando usamos la palabra reservada “type” para crear un nuevo alias de tipo podríamos pensar que literalmente estamos creando un tipo nuevo, pero la realidad es que estamos creando un nuevo nombre para un tipo. Por lo cual a la hora de la verdad “type” es una definición de un tipo de dato como por ejemplo “tuple”, “string”, etc.

Sin embargo, crear estos alias de tipos puede ser muy útil para simplificar algunas cosas, por ejemplo cuando necesitamos crear funciones que retornen un Object que nosotros creamos. En ese caso estos alias de tipo son convenientes porque podemos hacer por ejemplo:

```
type Person = {
  name: string,
  age: number
};

type ReturnPerson = (
  person: Person
) => Person;

const returnPerson: ReturnPerson = (person) => {
  return person;
};
```

Observamos que al hacer esto podemos referir los types como “los tipos” de objeto que esperamos, por lo cual es una manera en la que al momento de implementar podríamos ver y referir más claramente a un tipo creado por nosotros. Pero es importante no perder de vista lo que realmente ocurre por debajo.

ii. Describa el funcionamiento del sistema de tipos del lenguaje, incluyendo el tipo de equivalencia para sus tipos, reglas de compatibilidad y capacidades de inferencia de tipos.

Para hablar del **sistema de tipos de TypeScript**.

Primero tenemos que **“tu JavaScript es TypeScript”.** Esto es porque TS proporciona seguridad de tipo en tiempo de compilación para código en JS. Esto no es sorpresa si pensamos en su nombre pero lo bueno es que los tipos son completamente opcionales, es decir que un archivo “.js” se puede renombrar a un archivo “.ts” y TS aún devolverá un “.js” válidop

equivalente al archivo JS original. Esto ocurre porque TS es intencional y estrictamente un superconjunto de JS con verificación de tipo opcional.

Por otra parte **“los tipos pueden ser implícitos”**, es decir, tenemos “inferencia de tipos”. TS intentará inferir tanta información de tipo como sea posible para brindar seguridad de tipo con un costo mínimo de productividad durante el desarrollo del código. Por ejemplo:

```
var foo = 123;
foo = '456'; // Error: cannot assign `string` to `number`
// foo es un entero o un string?
```

En este ejemplo cuando primero le asignamos el número “123” a “foo”, TS inmediatamente infirió entonces que el tipo de “foo” es “number”, de esta manera cuando luego intentamos asignarle un string, este nos da un error.

Esta forma de inferir tiene una buena motivación ya que busca cuidar que el programador no haga cosas locas como las que pretendía el ejemplo, puesto que si se hubiese podido entonces en el resto del código no íbamos a poder estar seguros si foo era una cadena o un string (al menos no sin estar chequeando cada cierto tiempo).

Por otra parte **“los tipos pueden ser explícitos”**, ya vimos que TS inferirá todo lo que pueda de forma segura, sin embargo se pueden usar “**anotaciones de tipo”** para ayudarnos a que lo que vea el compilador sea lo que pensamos que debería ver indicándole de manera directa por medio de anotación de tipo postfija el tipo de alguna variable (es decir, que le especificamos a TS el tipo de dato con el que trabajaremos sobre una variable). Por ejemplo:

```
var foo: number = 123;
```

De esta manera si nos equivocamos el compilador reportará un error, por ejemplo:

```
var foo: number = '123'; // Error: cannot assign a `string` to a `number`
```

También temenos que **“los tipos son estructurales”**. En algunos lenguajes la escritura estática es innecesariamente ceremoniosa, porque aunque sabemos que un código funcionará bien, la semántica del lenguaje nos obliga a copiar cosas. En TS los tipos son estructurales, esto significa que “duckt typing” es una construcción de lenguaje de primera clase. Por ejemplo, si tenemos:

```
interface Point2D {
    x: number;
    y: number;
}
interface Point3D {
    x: number;
```

```
    y: number;
    z: number;
}
var point2D: Point2D = { x: 0, y: 10 }
var point3D: Point3D = { x: 0, y: 10, z: 20 }

function iTakePoint2D(point: Point2D) { /* do something */ }
iTakePoint2D(point2D); // exact match okay
iTakePoint2D(point3D); // extra information okay
iTakePoint2D({ x: 0 }); // Error: missing information `y`
```

La función iTakePoint2D aceptará cualquier cosa que contenga todas las cosas “x” e “y” que espera.

Otra característica que tiene el sistema de tipos es que **“no impide la emisión de JavaScript”**, esto es que para facilitar la migración de código JS a TS, incluso si hay errores de compilación, de forma predeterminada, TS emitirá JS válido lo mejor que pueda. Por ejemplo:

```
var foo = 123;
foo = '456'; // Error: cannot assign a `string` to a `number`
```

Este Código emitirá el JS:

```
var foo = 123;
foo = '456';
```

De esta manera en forma gradual es posible actualizar el código JS a TS. Cabe destacar que esto es muy diferente a como funcionan compiladores de otros lenguajes.

Otra característica del sistema de tipos es que **“los tipos pueden ser *ambient*”**. Como uno de los principales objetivos de TS era hacer posible el uso seguro de las bibliotecas de JS existentes, éste se hace cargo de eso mediante la declaración. De esta manera, TS proporciona una escala variable de cuánto a cuánto esfuerzo deseamos poner en las declaraciones (cuanto más esfuerzo, más seguridad de tipos + inteligencia de código). Hay que tener en cuenta que la comunidad “DefinitelyTyped” ha escrito las definiciones para la mayoría de las bibliotecas populares de JS, por lo que para la mayoría de propósitos el archivo de definición ya existe o al menos tiene una amplia lista de plantillas de declaración de TS bien revisadas que están disponibles.

Como ejemplo de esto, si quisiéramos crear nuestro propio archivo de declaración, vamos a considerar un caso con “jquery”.

```
$('.awesome').show(); // Error: cannot find name `$`
```

Por defecto TS espera a que declaremos, es decir, que usemos “var” en algún sitio antes de usar una variable, entonces, como solución rápida puede decirle a TS que de hecho hay algo llamado “$”, de la siguiente manera:

```
declare var $: any;
$('.awesome').show(); // Okay!
```

Pero si quisiéramos, podríamos proporcionarle más información a esta definición básica para ayudarle a protegerse de errores, entonces podríamos tener:

```
declare var $: {
(selector:string): any;
};
$('.awesome').show(); // Okay!
$(123).show(); // Error: selector needs to be a string
```

Es importante tener en cuenta que el sistema de tipos de TS realiza como tal una formalización de los tipos de JS mediante una representación “estática” de sus tipos dinámicos. Esto permite entonces que como desarrolladores podamos definir variables y funciones tipadas sin perder la “esencia” de JS. Sin embargo, poder definir los tipos durante el tiempo de diseño nos ayuda a evitar errores en tiempo de ejecución, como por ejemplo pasar el tipo de variable incorrecto a una función.

Entonces, aunque no sea obligatorio en TS definir el tipo de dato de una variable, tiene su beneficio ya que nos permitirá realizar escritura segura tanto en TS como en JS una vez que se haga la transpilación)

Trabajar con tipos de datos estáticos nos permitirá verificar en tiempo real de desarrollo si el dato que introducimos es correcto o no ya que el compilador analiza los tipos y en caso de detectar un error, nos mostrará dio error por pantalla.

Como vimos anteriormente, podremos contar con inferencia de tipos (TS asignará un tipo de dato a una variable dependiendo del valor que le asignemos inicialmente a dicha variable) y esta puede realizarse de dos maneras:

1. **Inferencia de Tipos (sin definir un tipo de dato):** Donde establecerá el dato como tipo “any”, es decir, que cualquier tipo de dato será válido. Esto si por ejemplo escribimos:

   ```
   let numero;
   ```

2. **Inferencia de tipos con valor:** Donde establecerá el tipo de dato sobre el valor que asignemos a dicha variable (haciendo un “typeOf” para obtener el tipo). Esto es lo que vimos antes que es cuando tenemos, por ejemplo:

```
let num2 = 42;
```

Ahora, si hablamos de la **compatibilidad de los tipos**, esto es lo que me determinará si algo de un tipo podrá ser asignado a otro. Por ejemplo:

```
let str: string = "Hello";
let num: number = 123;
str = num; // ERROR: `number` is not assignable to `string`
num = str; // ERROR: `string` is not assignable to `number`
```

Acá Podemos ver que String y Numer no son compatibles.
Entre algunas de las características más resaltantes de compatibilidad de TS tenemos:

Primero que nada **"Soundness". E**l sistema de tipos de TS esta diseñado para ser conveniente, por lo cual permite comportamientos "poco sólidos" (unsound). Por ejemplo:

```
let foo: any = 123;
foo = "Hello";
// Later
foo.toPrecision(3); // Allowed as you typed it as `any`
```

En este caso, por cómo se comporta el tipo "any" entonces prácticamente le decimos al compilador que haga lo que quiera, puesto que puede asignar "cualquier cosa" a una variable tipo "any". Aquí vemos como primero le asignamos un number y luego un string y no pasó nada.

Por otro lado, tenemos **compatibilidad estructural**. Los objetos en TS están tipados estructuralmente. Esto significa que los "nombres" no importan siempre y cuando las estructuras coincidan (hagan "match"). Entonces, por ejemplo:

```
interface Point {
    x: number,
    y: number
}

class Point2D {
    constructor(public x:number, public y:number){}
}

let p: Point;
// OK, because of structural typing
```

p = new Point2D(1,2);

Esto nos permite crear objetos “on the fly” y aún tener seguridad cuando tenga que ser inferido.

Otro aspecto de la compatibilidad es la **“Variance” o varianza**, e un concepto importante y fácil de entender para el análisis de compatibilidad de tipos. Por ejemplo, si tenemos los tipos simples “Base” y “Child”. Si Child es un hijo de Base, las instancias de Child se pueden asignar a una variable de tipo Base (polimorfismo *wink*).

En cuanto a la compatibilidad de tipos, de tipos complejos compuestos de esos mismos tipos Base y Child, todo dependerá del escenario en el que se encuentren “Base” y “Child”. La varianza puede ser:

- **Convariante (Covariant):** Solo varianza en una dirección por ejemplo de A a B.
- **Contravariante (Contravariant):** Solo en dirección opuesta. Es decir que para el ejemplo puesto antes entonces sería de B a A.
- **Bivariante (Bivariant):** tanto covariante como contravariante. Es decir de A a B y de B a A.
- **Invariante (Invariant):** Si los tipos no son exactamente iguales entonces son incompatibles.

Es importante destacar que para un sistema de tipos completamente sólido (sound) en presencia de datos mutables como JS, “invariante” es la única opción válida para establecer comparación. Sin embargo, en TS “la conveniencia” que nos brinda, esa libertad nos obliga a que sea posible tomar decisiones incorrecta considerando las otras opciones.

Cuando comparamos funciones debemos considerar:

Primero el **tipo de retorno.** Aquí utilizamos la “covarianza” (el tipo de retorno debe contener al menos suficiente data). Por ejemplo:

```
/** Type Hierarchy */
interface Point2D { x: number; y: number; }
interface Point3D { x: number; y: number; z: number; }
/** Two sample functions */
let iMakePoint2D = (): Point2D => ({ x: 0, y: 0 });
let iMakePoint3D = (): Point3D => ({ x: 0, y: 0, z: 0 });
/** Assignment */
iMakePoint2D = iMakePoint3D; // Okay
iMakePoint3D = iMakePoint2D; // ERROR: Point2D is not assignable to Point3D
```

Otro aspecto para considerar es el “número de argumentos” ya que después de todo, está garantizado que “serás llamado con al menos suficientes argumentos” (ya que las funciones pudiesen ignorar parámetros adicionales). Por ejemplo:

```
let iTakeSomethingAndPassItAnErr
= (x: (err: Error, data: any) => void) => { /* do something */ };
iTakeSomethingAndPassItAnErr(() => null) // Okay
iTakeSomethingAndPassItAnErr((err) => null) // Okay
iTakeSomethingAndPassItAnErr((err, data) => null) // Okay
// ERROR: Argument of type '(err: any, data: any, more: any) => null' is not assignable to
parameter of type '(err: Error, data: any) => void'.
iTakeSomethingAndPassItAnErr((err, data, more) => null);
```

También se considera el **“tipo de los argumentos”.** Aquí se utiliza la varianza “bivariante” (diseñada para soportar el manejo de escenarios de eventos comunes). Por ejemplo:

```
/** Event Hierarchy */
interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

/** Sample event listener */
enum EventType { Mouse, Keyboard }
function addEventListener(eventType: EventType, handler: (n: Event) => void) {
/* ... */
}

// Unsound, but useful and common. Works as function argument comparison is bivariant
addEventListener(EventType.Mouse, (e: MouseEvent) => console.log(e.x + "," + e.y));

// Undesirable alternatives in presence of soundness
addEventListener(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + "," +
(<MouseEvent>e).y));
addEventListener(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.log(e.x +
"," + e.y)));

// Still disallowed (clear error). Type safety enforced for wholly incompatible types
addEventListener(EventType.Mouse, (e: number) => console.log(e));
```

´Por otro lado como un ejemplo de la covarianza en acción, tenemos que “Array<Child>” es asignable a “Array<Base>” si las funciones son compatibles. Y para esto (que se cumpla la covarianza) requerimos que todas las funciones de “Array<Child>” sean asignables a

“Array<Base>”. Por ejemplo, que “push(t:child) sea asignable a push(t:Base)” lo cual se hace posible por bivarianza en el argumento de la función. Un ejemplo que hace referencia a esto sería:

```
/** Type Hierarchy */
interface Point2D { x: number; y: number; }
interface Point3D { x: number; y: number; z: number; }

/** Two sample functions */
let iTakePoint2D = (point: Point2D) => { /* do something */ }
let iTakePoint3D = (point: Point3D) => { /* do something */ }
iTakePoint3D = iTakePoint2D; // Okay : Reasonable
iTakePoint2D = iTakePoint3D; // Okay : WHAT
```

En esas ultima parte todos esperamos que ocurra un error, sobretodo en “iTakePoint2D = iTakePoint3D;” pero nope, no en TS.

En el caso de compatibilidad entre clases, solo se comparan los miembros y métodos de la instancia. Los constructores y la estáticos no juegan ningún papel. Por ejemplo:

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { /** do something */ }
}
class Size {
    feet: number;
    constructor(meters: number) { /** do something */ }
}
let a: Animal;
let s: Size;
a = s; // OK
s = a; // OK
```

Por otro lado, los miembros privados y protegidos deben provenir de la misma clase. Dichos miembros esencialmente hacen que la clase sea nominal. Por ejemplo:

```
/** A class hierarchy */
class Animal { protected feet: number; }
class Cat extends Animal { }
let animal: Animal;
let cat: Cat;
animal = cat; // OKAY
cat = animal; // OKAY
```

```
/** Looks just like Animal */
class Size { protected feet: number; }
let size: Size;
animal = size; // ERROR
size = animal; // ERROR
```

También hay 2 tipos de compatibilidad importantes dentro de TS y se llaman “subtype” y “assigment” para subtipos y asignación. Éstas solo difieren en que la asignación extiende la compatibilidad de subtipos con las reglas para permitir la asignación “hacia y desde cualquier” y “hacia y desde enum” con los valores numéricos correspondientes.

En diferentes sitios del lenguaje se utilizan uno de estos mecanismos de compatibilidad según la situación, pero a efectos prácticos la compatibilidad de tipos viene dictada mayormente por la compatibilidad de asignaciones, incluso en los casos de las cláusulas “implements” y “extends”.

En este caso, en la siguiente tabla podemos ver resumido “quién se puede asignar con quién” entre algunos tipos abstractos:

	any	unknown	object	void	undefined	null	never
any →		✓	✓	✓	✓	✓	✗
unknown →	✓		✗	✗	✗	✗	✗
object →	✓	✓		✗	✗	✗	✗
void →	✓	✓	✗		✗	✗	✗
undefined →	✓	✓	✓	✓		✓	✗
null →	✓	✓	✓	✓	✓		✗
never →	✓	✓	✓	✓	✓	✓	

Algunas cosas que debemos considerar dentro de TS son:

- Todo es asignable a si mismo.
- Any y el tipo especial “unknown” son lo mismo en términos de lo que se les puede asignar pero diferentes en que “unknown” no es asignable a nada excepto “any”.
- “unknown” y “never” son inversos entre si. Todo es asignable a el primero el segundo es asignable a todo.
- “nothing” es asignable a “never” y “unknown” no es assignable a nada que no sea “any”.

- “void” no se puede asignar a nada ni desde nada con las excepciones de “any”, “unknown”, “never”, “undefined” y “null” si el indicador strictNullChecks esta desactivado.
- Cuando strictNullChecks esta desactivado “null” y “undefined” son similares a “never”: asignables a la mayoría de los tipos y la mayoría de los tipos no se les puede asignar. Además, son asignables entre si.
- Cuando strictNullChecks esta activado, “null” e “undefined” se comportan más como “void”: no asignable a nada ni desde nada, excepto “any”, “unknown”, “never” y “void” (“undefined” siempre se puede asignar a “void”).

Finalmente, si hablamos de **equivalencia para los tipos** de TS tenemos:

La **igualdad estricta (===),** con la cual si tenemos x **===** y, “x” e “y” son valores y producirán “true” solo si los tipos de “x” e “y” son iguales y si los valores de “x” e “y” son iguales, de lo contrario nos retornara “false”.

Cuando comparamos la igualdad de dos valores de esta manera, ninguno de estos valores se convierte de manera implícita antes de ser comparado. Si los valores tienen tipos diferentes son considerados diferentes. Por el contrario, si los valores tienen el mismo tipo y no son números, son considerados iguales si tienen el mismo valor. Finalmente si ambos valores son números, son considerados iguales si ambos no son NaN y tienen el mismo valor o si uno es +0 y el otro -0.

Por ejemplo:

```
var num = 0;
var obj = new String("0");
var str = "0";
var b = false;

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

Por otro lado temenos la **igualdad débil (**==**),** con la cual si tenemos x **==** y, “x” e “y” son valores y producirán “true” o “false”. Aquí es importante saber que al comparar de esta manera JS en tiempo de ejecución realizará conversiones de tipos para que ambos valores sean del mismo tipo. Entonces puede ocurrir, por ejemplo:

```
let a = 10;

a == 10          //true
a == '10'        //true
```

En este caso vemos que a pesar de que a es un “number” y “10” es un string, el resultado de la igualdad débil nos arrojará “true”. Y esto ocurre porque la expresión:

```
"10" == 10
```

En tiempo de ejecución JS ejecuta con una conversión de tipos, por lo cual realmente se hace lo siguiente:

```
ToNumber("10") === 10
```

Podemos ver que tras la conversión, la comparación que se lleva a cabo funciona exactamente como “===”. La igualdad débil es una igualdad simétrica: A == B tiene una semática idéntica a B == A para cualquier valor que tengan A y B (excepto por el orden de las conversiones de tipo aplicadas, que como vimos se hace sobre el elemento de la izquierda).

Finalmente existe otro tipo de equivalencia llamada **“igualdad Same-Value”**, esta se encarga de un último caso de uso que es determinar si dos valores son funcionalmente idénticos en todos los contextos. Un ejemplo de esto es cuando se intenta hacer mutable una propiedad inmutable:

```
// Add an immutable NEGATIVE_ZERO property to the Number constructor.
Object.defineProperty(Number, "NEGATIVE_ZERO",
                { value: -0, writable: false, configurable: false, enumerable: false });

function attemptMutation(v)
{
  Object.defineProperty(Number, "NEGATIVE_ZERO", { value: v });
}
```

“Object.defineProperty”, que lanzará una excepción cuando se intente cambiar una propiedad inmutable, la cambiará pero no hará nada si al solicitar el cambio actual: Si “v” es “-0”, no ha sido solicitado nindún cambio y no se lanzará ningún error. Pero si “v” es “+0”, Number. NEGATIVE_ZERO no tendrá más su valor inmutable. Internamente al redefinir una

propiedad inmutable, el nuevo valor se compara con el valor actual usando la igualdad same-value.

Como tal el método El método “Object.is” nos proporciona la igualdad same-value.

A continuación, tenemos una imagen que resume las equivalencias entre algunos tipos:

x	y	==	===	Object.is
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
"foo"	"foo"	true	true	true
{ foo: "bar" }	x	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
"0"	0	true	false	false
"17"	17	true	false	false
[1,2]	"1,2"	true	false	false
new String("foo")	"foo"	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: "bar" }	{ foo: "bar" }	false	false	false
new String("foo")	new String("foo")	false	false	false
0	null	false	false	false
0	NaN	false	false	false
"foo"	NaN	false	false	false
NaN	NaN	false	false	true

iii. Explique los diferentes tipos de rutinas que el lenguaje ofrezca, así como los diferentes tipos de pasaje de parámetros.

TS permite aportar varias características de las cuales JS no dispone al momento de plantear funciones y métodos.

En primer lugar tenemos **"Parámetros tipados y funciones que retornan un valor".** Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso de que estemos en una clase. Por ejemplo:

```
function sumar(valor1:number, valor2:number): number {
  return valor1+valor2;
}

console.log(sumar(10, 5));
```

En el ejemplo tenemos resaltado en **negritas** la forma en la que indicamos el tipo de los argumentos y el tipo de retorno de la función como mencionamos antes. La función suma entonces recibe dos parámetros de tipo number y retorna a su vez un valor de tipo number.

Si llamamos a esta función intentando pasarle un valor de tipo distinto nos arrojará error (que intentemos por ejemplo: "console.log(sumar('juan', 'carlos'));")**.** Lo mismo ocurrirá si la ponemos a retornar algo que no sea del tipo number (por ejemplo que suma retornara por alguna razón "return 'Hola mundo';"). El tipado estático que ofrece TS favorece a identificar este tipo de errores antes de ejecutar la aplicación.

Otra cosa que ofrece TS son **las funciones anónimas**, que son tales que no especifican un nombre. Estas son semejantes a JS con la salvedad de la definición de tipos de los parámetros. Por ejemplo:

```
const funcSumar = function (valor1:number, valor2:number): number {
  return valor1 + valor2;
}

console.log(funcSumar(4, 9));
```

También TS ofrece para las funciones, rutinas y métodos **"Parámetros opcionales"**. Podremos agregar el carácter '?' al nombre del parámetro para indicar que podría o no llegar un dato ahí. Por ejemplo:

```
function sumar(valor1:number, valor2:number, valor3?:number):number {
  if (valor3)
    return valor1+valor2+valor3;
  else
    return valor1+valor2;
```

```
}

console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

Este ejemplo nos indica que el tercer parámetro de la función es opcional. Entonces podremos llamar a la función “sumar” pasando 2 o 3 valores numéricos, haciendo por ejemplo:

```
console.log(sumar(5,4));    //Obtenemos 9
console.log(sumar(5,4,3));  //Obtenemos 12
```

En caso de que pasemos una cantidad de parámetros distinta a 2 o 3 se generará un error en tiempo de compilación. Por ejemplo, si intentamos pasar cuatro argumentos obtendremos “error TS2554: Expected 2-3 arguments, but got 4”.

Adicionalmente se pueden tener tantos parámetros opcionales como se necesiten, pero es importante que estos sean los últimos parámetros definidos en la función.

También podemos tener **“Parámetros por defecto”.** Es una característica de TS que nos permite asignar un valor por defecto a un parámetro para los casos en que la llamada a la misma no se le envíe algo. Por ejemplo:

```
function sumar(valor1:number, valor2:number, valor3:number=0):number {
  return valor1+valor2+valor3;
}

console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

En el ejemplo el tercer parámetro almacena un cero si no le pasamos al gún otro valor numérico en la llamada a la función como hacemos en “console.log(sumar(5,4));”.

En el caso de estos valores por defecto, también podemos tener varios pero deben ser los últimos, es decir que primero indicamos aquellos parámetros que reciben datos de forma obligatoria cuando los llamamos y finalmente indicamos aquellos que tienen valores por defecto.

Otra característica que posee TS es la de permitirle a las funciones **“parámetros Rest”.** Esto es la posibilidad de pasar una lista indefinida de valores y que los reciba un vector. Esto se logra antecediendo tres puntos al nombre del parámetro, por ejemplo:

```
function sumar(...valores:number[]) {
  let suma=0;
  for(let x=0;x<valores.length;x++)
```

```
   suma+=valores[x];
  return suma;
}

console.log(sumar(10, 2, 44, 3));
console.log(sumar(1, 2));
console.log(sumar());
```

El parámetro “valores” se le anteceden los tres puntos seguidos e indicamos que se trata de un vector de tipo “number”. Luego, cuando llamamos a la función le pasamos una lista de valores enteros que luego la función empaquetará en el vector.

Las funciones con un parámetro Rest pueden tener otros parámetros pero nuevamente deben declararse antes que este y es importante saber que los parámetros Rest no pueden tener valores por defecto.

Otra opción que TS nos da para las funciones es **“el operador Spread”**, este permite descomponer una estructura de datos en elementos individuales. Es la operación inversa de los parámetros Rest. La sintáxis se aplica anteponiendo al nombre de la variable tres puntos. Por ejemplo:

```
function sumar(valor1:number, valor2:number, valor3:number):number {
  return valor1+valor2+valor3;
}

const vec:number[]=[10,20,30];
const s=sumar(...vec);
console.log(s);
```

Observamos que en este caso lo usamos al momento de llamar la función como en “s=sumar(...vec);” y en este caso el vector “vec” será descompuesto en sus distintos valores, para ser asignados como los 3 argumentos de la función en orden.

TS también ofrece las llamadas **“funciones callbacks”**. Estas se pueden pasar a otra función como parámetro y dentro de esa función ser llamadas. Por ejemplo:

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number):
number {
  return func(valor1, valor2);
}

console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))
```

```
console.log(operar(3, 7, (x: number,y: number): number => {
  return x-y;
}))

console.log(operar(3, 7, (x: number,y: number): number => {
  return x*y;
}))
```

Vemos que la función “operar” recibe tres parámetros donde el tercero es de tipo función que recibe dos parámetros number y retorna un number. Podemos ver que en el ejemplo la llamamos tres veces y le pasamos funciones que procesan dos enteros para obtener su suma, resta y multiplicación respectivamente.

Para hacer más claro el código, podemos usar la palabra clave “type” que vimos mucho más arriba que nos permite hacer las veces de “crear nuevos tipos” para luego reutilizarlos. Entonces podríamos tener, por ejemplo:

```
type Operacion=(x: number, y:number)=>number;

function operar(valor1: number, valor2: number, func: Operacion): number {
  return func(valor1, valor2);
}
```

El “type” operación tiene la misma firma de una función con dos parámetros de tipo number y el retorno number como teníamos inicialmente. Podemos usarlo para indicar el tipo de la función entonces.

Otra cosa que nos brinda TS para funciones son los **parámetros de tipo unión**. En estos usamos la misma sintaxis del tipo unión para variables pero en los parámetros de una función, por ejemplo:

```
function sumar(valor1: number | string, valor2: number | string ): number | string  {
  if (typeof valor1 ==='number' && typeof valor2 ==='number')
    return valor1+valor2;
  else
    return valor1.toString() + valor2.toString();
}

console.log(sumar(4, 5));
console.log(sumar('Hola ', 2));
console.log(sumar('Hola ', 'Mundo'));
```

En este caso con condicionales identificamos que operación a realizar según los tipos de datos de los parámetros. En el ejemplo si los dos parámetros son enteros se reciben datos de tipo number y podemos sumarlos como enteros. Pero en caso contrario, si alguno es string entonces pasamos ambos a string (por si acaso solo uno es string y el otro no).

Hasta ahora hemos visto como funcionan estas cosas para las funciones, sin embargo podemos destacar que TS permite que todos estos conceptos puedan ser aplicados a métodos dentro de las clases. Por ejemplo:

```
class Operacion {
  sumar(...valores:number[]) {
    let suma=0;
    for(let x=0;x<valores.length;x++)
      suma+=valores[x];
    return suma;
  }
}

let op: Operacion;
op=new Operacion();
console.log(op.sumar(1,2,3));
```

Si hablamos de tipos de funciones en general TS posee:

- Funciones con nombres: Éstas en cierta forma las mencionamos de alguna manera antes, se diferencian de las de JS en que los argumentos pueden asignársele tipos y también a lo que devuelve la función. Por ejemplo:

```
function foo(msg: string) { console.log(msg); }
```

- Funciones anónimas: Estas las mencionamos antes de manera explícita. Por ejemplo:

```
var foo = function(x: number, y: number): number { return x+y; }
```

- Funciones Lambda: Éstas son funciones de una sola instrucción normalmente que se almacenan en una variable o que se ejecutan directamente. Se caracterizan porque no hace falta escribir return ya que se sobreentiende que la función devuelve algo. Por ejemplo:

Normalmente lo común es escribir:

```
var foo = function (x, y) { return x + y; };
```

En la forma lambda sería, por ejemplo:
```
var foo = (x: number, y: number) => x+y;
```

Se caracterizan por el "=>". Cabe destacar que no son funciones excusivas de TS ya que C# y Python también las tienen pero son una opción a utilizar (y hasta pueden llegar a escribirse de una forma más declarativa).

- Funciones de clases (métodos): La peculiaridad que tienen es que no aparece "fuction" sino que se escriben directamente dentro de las clases. Por ejemplo:

```
class NewClass {

        foo(x: number, y:number): number { return x+y; }

}
```

En cuanto a las **corutinas**, técnicamente estas no existen, aquí pero TS tiene un sistema llamado "Promises" con el que en cierta forma se puede modelar lo que vendrían siendo las corrutinas. Para esto escribimos, por ejemplo:

```
export function waitforseconds(seconds: number): Promise<void>
{
   return new Promise<void>(resolve =>
   {
      setTimeout(() => { resolve(); }, seconds * 1000);
   });
}
```

Creamos una función "waitForSeconds" con el foco de que esta se comporte como las que usamos en Unity o la equivalente en Unreal. Una vez constuida de esta forma, podemos agregar "waitForSeconds" en cualquier función asíncrona de la siguiente forma:

```
async codeExample() : Promise<void>
   {
      for (var i = 0; i < 5; i++)
      {
         cc.log("You'd put any code you wanted to happen before the next iteration here.");
         await waitforseconds(1.1); //And this will wait for 1.1 seconds.
      }
   }
```

Esta forma, a diferencia de las otras funciones de clases, comienza con "async" antes del nombre de la función. Esto es necesario para que "await" funcione. Luego tenemos el tipo de función "Promise<void>" en este caso void porque no retornamos nada.

Luego si queremos llamarla lo hacemos como a cualquier otra función, en este caso "this.codeExample();" funcionará sin problemas. Entonces al programar un código podríamos usar tantos "awaits" como queramos en cualquier sitio. La función esperará por la duración colocada a que pase antes de continuar con lo lógica pero sin colgar o detener el programa en ese tiempo ya que funcionará como una corutina que se ejecuta a la par (o varias corutinas si la iniciamos en distintos puntos, y ojo que cada await será independiente).

Cabe destacar que esta no es la ultima manera de construir corutinas, investigando mas en las profundidades de internet pueden encontrarse otras implementaciones alternas.

Finalmente, en cuanto al **pasaje de parámetros por valor y por referencia** es importante saber que en TS cuando asignamos un valor a una variable, o pasamos un argumento a una función, este proceso siempre se hace por valor. Como trabajamos con características de JS es importante saber que este no nos ofrece la opción de pasar o asignar "por referencia" como en otros lenguajes. Sin embargo, hay algo particular que pasa en este lenguaje y es que cuando una variable hace referencia a un "Objeto" (Object, Array o Function), el "valor" es la referencia en si.

Cuando asignamos por valores "primitivos" (Boolean, Null, Indefined, Number, String y Symbol), el valor asignado es una copia del valor que estamos asignando.

Pero cuando asignamos valores no primitivos o complejos (Object, Array y Function) entonces TS copia la "referencia", lo que implica que no se copia el valor en si, sino una referencia a través de la cual accederemos al valor original.

Es como un pseudo pase por referencia. Por esto podemos encontrarnos que cuando alguien quiere "pasar valores por referencia", como esto vemos que en TS no se puede hacer a priori por como esta todo definido, entonces la gente lo que hace es "envolver" los valores metiéndolos en un objeto. Por ejemplo:

Si inicialmente se tenía:

```
function foo(value1: number, value2: number) {
    value1++;
    value2++;
 }
```

Pero se quieren pasar "por referencia" los valores entonces hacen algo como esto:

```
function foo(model: {property1: number; property2: number}) {
    model.property1++;
    model.property2++;
    // No se necesita pero se considera buena practica.
    return model;
}
```

```
const bar = { property1: 0, property2: 1 };
foo(bar);

console.log(bar.property1) // 1
console.log(bar.property2) // 2
```

Donde vemos que pasan los valores envueltos en un objeto para simular entonces un pasaje por referencia. Sin embargo, sabemos entonces que realmente el mecanismo de pasaje de parámetros de TS es pasaje por valor en general y legalmente en vez de un pasaje por referencia, utilizan la trampa del pasaje por "sharing".

iv. Detalle el mecanismo de manejo de excepciones del lenguaje. Si no tiene uno, explique cómo un programador usual del mismo haría para tener una alternativa a esto (por ejemplo, la pareja setjmp/longjmp de C).

En cuanto al mecanismo de manejo de excepciones, TS aprovecha la clase Error de JS para utilizarla en las excepciones. Podemos lanzar un error con la palabra clave "throw" y atraparlo con un bloque "try/catch". Por ejemplo:

```
try {
        throw new Error('Something bad happened');
}
catch(e) {
        console.log(e);
}
```

Dependiendo del tipo de error, es posible utilizar las propiedades name y message para obtener un mensaje más refinado.

La propiedad name proporciona la clase general de Error (tal como DOMException o Error), mientras que message generalmente proporciona un mensaje más conciso que el que se obtendría al convertir el objeto error en una cadena.

Si lanzamos nuestras propias excepciones (que también es una opción que tenemos), para aprovechar estas propiedades (por ejemplo, si nuestro bloque catch no discrimina entre nuestras propias excepciones y las del sistema), podemos usar el constructor Error. Por ejemplo:

```
function doSomethingErrorProne() {
  if (ourCodeMakesAMistake()) {
    throw (new Error('El mensaje'));
  } else {
    doSomethingToGetAJavascriptError();
  }
}
```

```
:
try {
  doSomethingErrorProne();
} catch (e) {            // AHORA, en realidad usamos `console.error()`
  console.error(e.name);   // registra 'Error'
  console.error(e.message); // registra 'The message' o un mensaje de error de JavaScript
}
```

De hecho, hasta podríamos no lanzar un error con “throw”. Tenemos la opción de simplemente pasar un objeto de la clase Error tomando una función callback con el primer argumento como un objeto error. Por ejemplo:

```
function myFunction (callback: (e?: Error)) {
        doSomethingAsync(function () {
                if (somethingWrong) {
                        callback(new Error('This is my error'))
                } else {
                        callback();
                }
        });
}
```

Más allá de la clase Error que tenemos incorporada, hay algunas clases de error, también incorporadas, adicionales que heredan de Error y que pueden generarse en tiempo de ejecución en JS:

-RangeError: Crea una instancia representando un error que ocurre cuando la variable numérica o parámetro esta fuera del rango válido. Por ejemplo

```
// Call console with too many arguments
console.log.apply(console, new Array(1000000000)); // RangeError: Invalid array length
```

-ReferenceError: Crea una instancia representando un error que ocurre cuando dereferenciamos una referencia inválida. Por ejemplo:

```
'use strict';
console.log(notValidVar); // ReferenceError: notValidVar is not defined
```

-**SyntaxError:** Crea una instancia representando un error de sintaxis que ocurre parseando código que no es válido en JS. Por ejemplo:

```
1***3; // SyntaxError: Unexpected token *
```

-**TypeError:** Crea una instancia representando un error que ocurre cuando la variable o parámetro no es de tipo válido. Por ejemplo:

```
('1.2').toPrecision(1); // TypeError: '1.2'.toPrecision is not a function
```

-URIError: Crea una instanci representando un error que ocurrió cuando se le paso parámetros inválidos a “encodeURI()” o a “decodeURI()”. Por ejemplo:

```
decodeURI('%'); // URIError: URI malformed
```

En cuanto a “throw”, se puede lanzar cualquier expresión, no solo de un tipo específico. Por ejemplo, podemos tener:

```
throw 'Error2';   // tipo String
throw 42;         // tipo Number
throw true;       // tipo Boolean
throw {toString: function() { return "¡Soy un objeto!"; } };
```

Podemos especificar un object cuando lanzamos una excepción y podemos hacer referencia a las propiedades del objeto en el bloque catch. Hacer algo por ejemplo:

```
// Crea un objeto tipo de UserException
function UserException(message) {
  this.message = message;
  this.name = 'UserException';
}

// Hacer que la excepción se convierta en una bonita cadena cuando se usa como cadena
// (por ejemplo, por la consola de errores)
UserException.prototype.toString = function() {
  return `${this.name}: "${this.message}"`;
}

// Crea una instancia del tipo de objeto y tírala
throw new UserException('Valor muy alto');
```

Como tal el bloque try /catch funciona como estamos acostumbrados. El bloque “try” contiene una o más declaraciones, y el bloque “catch” contiene declaraciones que especifican qué hacer si se lanza una excepción en el bloque try. Pero además podemos también contar de

manera opcional con el bloque “finally”, que se ejecutará después de que se ejecutan los bloques try y catch comportándose tal y como lo vimos en clase. Por ejemplo:

```
openMyFile();
try {
  writeMyFile(theData); // Esto puede arrojar un error
} catch(e) {
  handleError(e); // Si ocurrió un error, manéjalo
} finally {
  closeMyFile(); // Siempre cierra el recurso
}
```

En el ejemplo se abre un archivo y luego se ejecutan declaraciones que usan el archivo. Si se lanza una excepción mientras el archivo está abierto, el bloque finally cierra el archivo antes de que falle el script. Usar finally aquí *asegura* que el archivo nunca se deje abierto, incluso si ocurre un error.

Si el bloque finally devuelve un valor, este se convierte en el valor de rotorno de toda la producción “try…catch…finally” independientemente de las declaraciones return en los bloques ***try y catch. Por ejemplo:***

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch(e) {
    console.log(1);
    return true;    // esta declaración de retorno está suspendida
                    // hasta que el bloque finally se haya completado
    console.log(2); // no alcanzable
  } finally {
    console.log(3);
    return false;   // sobrescribe el "return" anterior
    console.log(4); // no alcanzable
  }
  // "return false" se ejecuta ahora
  console.log(5);   // inalcanzable
}
console.log(f()); // 0, 1, 3, false
```

Además, la sobreescritura de los valores devueltos por el bloque finally también se aplica a las excepciones lanzadas o relanzadas dentro del bloque catch. Por ejemplo:

```
function f() {
  try {
    throw 'bogus';
  } catch(e) {
    console.log('captura "falso" interno');
    throw e; // esta instrucción throw se suspende hasta
          // que el bloque finally se haya completado
  } finally {
    return false; // sobrescribe el "throw" anterior
  }
  // "return false" se ejecuta ahora
}

try {
  console.log(f());
} catch(e) {
  // ¡esto nunca se alcanza!
  // mientras se ejecuta f(), el bloque `finally` devuelve false,
  // que sobrescribe el `throw` dentro del `catch` anterior
  console.log('"falso" externo capturado');
}

// Produce
// "falso" interno capturado
// false
```

Por otra parte también podemos anidar una o más declaraciones try...catch.

Si un bloque try interno *no* tiene un bloque catch correspondiente, entonces ***debe*** contener un bloque finally, y el bloque catch adjunto de la declaración try...catch se comprueba para una coincidencia.

(b) Implemente los siguientes programas en el lenguaje escogido:

i. Defina un árbol binario con información en ramas y hojas y sobre este árbol, defina una función esDeBusqueda que diga si el árbol en cuestión es un árbol de búsqueda o no.

Esta en la carpeta de pregunta 1 del git: “arbolBinario.ts”

ii. Defina un tipo de datos recursivo que represente numerales de Church. Sobre este tipo se desea que implemente las funciones suma y multiplicación.

Esta en la carpeta de pregunta 1 del git: “church.ts”

Nota Final: Gracias por venir a mi TED Talk…
PD: AIURA, es demasiada información :’V