Bienvenides a fundamentos de programacion, capitulo 3!

- Diferencias entre const, let y var
- Modularización y funciones

Diferencias entre const, let y var

Antes de empezar a hablar de las diferencias de cada una tenemos que entender que es el ámbito o scope de una variable.

El scope de una variable se define como el alcance que tiene esta dentro del código, es decir, en que puntos del programa una variable puede ser usada y en qué puntos no.

Variable con Var

Las declaraciones **var** tienen alcance global y en la función alcance local.

El alcance es global cuando una variable `var` se declara fuera de una función. Esto significa que cualquier variable que se declare con `var` fuera de un bloque de funciones está disponible para su uso en todo el programa.

`var` tiene un alcance de función cuando se declara dentro de una función. Esto significa que está disponible y se puede acceder solo dentro de esa función.

Ejemplo

```
var variable_1 = "Esto es la variable 1";
function imprimir() {
  var variable_2 = "Esto es la variable 2";
}

console.log(variable_2); // error: varibale_2 is not defined
```

`variable_1` tiene un alcance global porque está declarada fuera de una función, mientras que `variable_2` tiene un alcance de función. Por lo tanto, no podemos acceder a la variable `variable_2` fuera de la función que pertenece y por eso nos devuelve un error.

Las variables declaradas con `var` **se pueden volver a declarar y actualizar,** esto significa que podemos hacer esto dentro del mismo alcance y no obtendremos un error.

```
var variable_1 = "Buenas! Soy la variable 1";

var variable_1 = "Chau! Soy la variable 1";

cosnole.log(variable_1); // Chau! Soy la variable 1
```

Que `var` sea de ámbito global tiene sus desventajas. Lo vemos en el siguiente ejemplo.

```
var variable_1 = "Buenas! Soy la variable 1";
if (true) {
  var variable_1 = "Chau! Soy la variable 1";
}

console.log(variable_1); // "Chau! Soy la variable 1"
```

Cuando dentro del `If` volvemos a declarar y asignarle un valor a `variable_1` pisamos a la primera declaración y asignación de `variable_1`. Si bien esto no es un problema si somos consientes de lo que esta pasando pero se convierte en un problema cuando no te das cuenta de que ya se ha definido un valor diferente en otra parte del código.

Esto puede causar muchos errores en tu código. Es por eso que let y const son necesarios.

Variable con let

El problema que comentábamos que tiene `var` lo soluciona `let`. El alcance de `let` es por bloque, un bloque es todo fragmento de código que este entre `{}`.

Por lo tanto, una variable declarada en un bloque con `let` solo está disponible para su uso dentro de ese bloque. Miremos el ejemplo anterior pero usando `let`.

```
let variable_1 = "Buenas! Soy la variable 1";
if (true) {
  let variable_1 = "Chau! Soy la variable 1";
  console.log(variable_1); // "Chau! Soy la variable 1"
}

console.log(variable_1); // "Buenas! Soy la variable 1"
```

Podemos observar que las declaraciones no se pisan, ya que cada variable vive en su bloque.

Si queremos acceder a una variable que existe en un ámbito diferente de donde fue declarada, nos devuelve un error. Esto se debe a que las variables son de ámbito de bloque.

```
let variable_1 = "Buenas! Soy la variable 1";
if (true) {
  let variable_2 = "Chau! Soy la variable 1";
  console.log(variable_2); // "Chau! Soy la variable 2"
}

console.log(variable_2); // variable_2 is not defined
```

A diferencia de `var`, una variable `let` no se puede volver a declarar dentro de su alcance, entonces si la volvemos a declarar nos va a devolver un error.

```
let variable_2 = "Primera vez";

let variable_2 = "Segunda vez";

// error: Identifier 'variable_2' has already been declared
```

Variable con const

Las variables declaradas con `const` tienen el mismo comportamiento que `let` con la diferencia que el valor se tiene que asignar al momento de la declaración y no puede cambiar. `const` no se puede actualizar o volver a declarar.

Miremos algunos ejemplos

```
const variable_1 = "Te saludo de nuevo!";
variable_1 = "Me voy";
// error: Assignment to constant variable.
```

Esto pasa porque al declararlo con `const` su valor no puede cambiar.

```
const variable_1 = "Te saludo de nuevo!";

const variable_1 = "Me voy";
// error: Identifier 'variable_1' has already been declared
```

Visto todo esto, mencionemos las diferencias entre `var`, `let` y `const`:

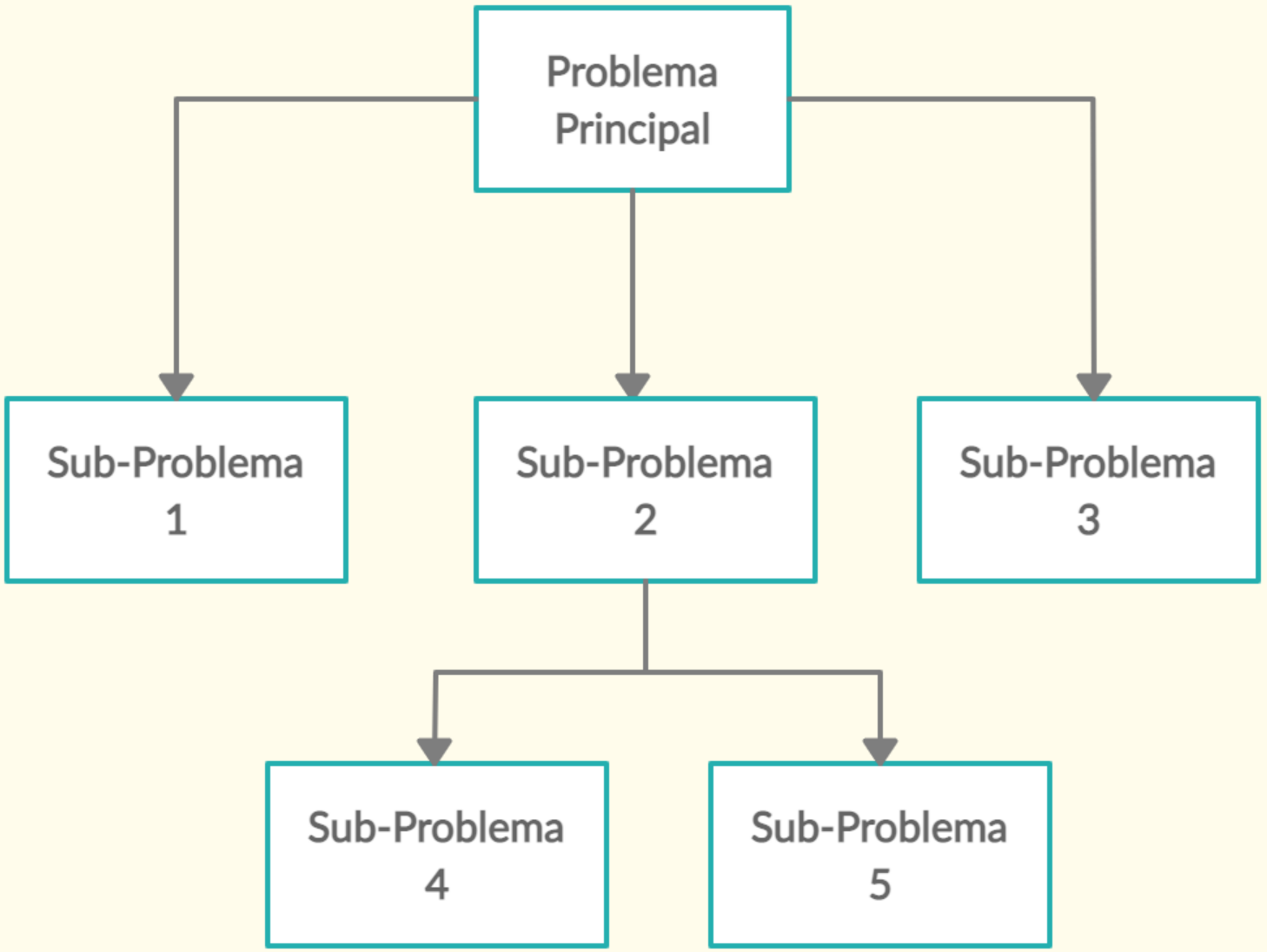
- Las declaraciones `var` tienen un alcance global o alcance a una función mientras que `let` y `const` tienen un alcance de bloque.
- Las variables `var` se pueden actualizar y volver a declarar dentro de su alcance, mientras que las variables `let` permite que las variables se puedan actualizar, pero no volver a declarar y las variables `const` no se pueden actualizar ni volver a declarar.
- Mientras que `var` y `let` pueden declararse sin inicializarse, const debe inicializarse durante la declaración.

Modularización y Funciones

Modularización

Dividir un problemas en partes mas simples.

Modularización quire decir dividir el programa en partes mas pequeña dedicadas a hacer un trabajo en especifico, la cual tendría que trabajar junto al resto del código. Los beneficios que nos da modularizar es no repetir código, fácil mantenimiento del programa y mejor legibilidad. La forma de modularizar en javascript es haciendo funciones.


Problema Principal
Sub-Problema 1
Sub-Problema 2
Sub-Problema 3
Sub-Problema 4
Sub-Problema 5

Funciones

Como hablamos anteriormente básicamente una función es un bloque de código encargado de una tarea en especifica. Estas funciones pueden recibir valores llamados parámetros.

"Una función en JavaScript es similar a un procedimiento — un conjunto de instrucciones que realiza una tarea o calcula un valor, pero para que un procedimiento califique como función, debe tomar alguna entrada y devolver una salida donde hay alguna relación obvia entre la entrada y la salida. Para usar una función, debes definirla en algún lugar del ámbito desde el que deseas llamarla."

Las funciones también pueden recibir datos como parámetros o no, los veremos un poco mas adelante.

Como definir funciones en javascript.

Una definición de función (también denominada declaración de función o expresión de función) consta de la palabra clave function, seguida de:

- El nombre de la función.
- Una lista de parámetros de la función, entre paréntesis y separados por comas. No necesariamente hay que pasarle parámetros.
- El bloque de código que vas a ejecutar para las funciones en JavaScript que definen entre llaves, `{ ... }`.
- Dentro de la función hay que agregar un return, es una palabra clave encargada de devolver "algo", el resultado de la ejecución del código.

Ejemplo:

```
function nombre(parametros_1, parametro_2){
//....
    bloque de codigo
...//
return valor_que_vas_a_devolver
}

function suma(numero_1, numero_2) {
  return numero_1 + numero_2;
}

console.log(suma(2, 3)); //5
```

Hablemos un poco de parametros en JavaScript

Primero vamos a decir que tenemos dos formas de pasar parámetros en programación en general, por valor y por referencia.

- **Por valor** significa que la función recibe sólo una copia del valor que tiene la variable, o sea que no la puede modificar.
- **Por referencia** significa que se pasa la posición de memoria donde esta guardada la variable, por lo que la función puede saber cuánto vale, pero además puede modificarla de cualquier manera.

Los parámetros primitivos **(Number, Boolean, String, Undefined)** en JavaScript se pasan por **valor** por eso mismo no podemos modificar esos parámetros dentro de la función para que se vean reflejados fuera.

Ejemplo por valor:

```
function nuevoValor(str) {
  str = "nuevo valor";
  return str;
}

const strValor = "Hola mundo";

console.log(nuevoValor(strValor)); // 'nuevo valor'

console.log(strValor); // 'Hola mundo'
```

Pero si pasamos un objeto como parámetro, este sí se va a modificar porque lo hace por referencia, es decir se van a ver reflejados fuera de la función.

Ejemplo por referencia:

```
function nuevoValor(obj) {
  obj.nuevo = "nuevo";
  return arr;
}
```

```
const obj = { valor_1: "valor_1", valor_2: "valor_2" };

nuevoValor(obj);

console.log(obj);
// {valor_1: 'valor_1', valor_2: 'valor_2', nuevo: 'nuevo'}
```

Cuando pasamos los parámetros hay que tomar en cuenta tres casos que se pueden presentar.

- Enviar menos parámetros de lo que tiene declarado la función, a esas variables que no le asignamos un valor javascript no dispara un error, sino que le asigna el valor **"undefined"**.

```
function foo(parametro_1, parametro_2, parametro_3) {
  console.log(parametro_1); // 1
  console.log(parametro_2); // 2
  console.log(parametro_3); // undefined
}

foo(1, 2);
```

- Enviar más parámetros de lo que definimos en la función, simplemente los ignora es decir no devuelve un error.

```
function foo(parametro_1, parametro_2, parametro_3) {
  console.log(parametro_1); // 1
  console.log(parametro_2); // 2
  console.log(parametro_3); // 3
}

foo(1, 2, 3, 6);
```

- El orden en que mandamos los parámetros y el que declaramos en la función tiene que ser el mismo y no necesariamente se tienen que llamar iguales.

```
function foo(parametro_1, parametro_2, parametro_3) {
  console.log(parametro_1); // 'parametro_1'
  console.log(parametro_2); // 'parametro_3'
  console.log(parametro_3); // ' parametro_2'
}

let parametro_1 = "parametro_1";
let parametro_2 = "parametro_2";
let parametro_3 = "parametro_3";

foo(parametro_1, parametro_3, parametro_2);
```

Llamar a las funciones

Definir una función no la ejecuta. Definirla simplemente nombra la función y especifica qué hacer cuando se llama a la función.

Básicamente llamar a la función consiste en escribir su nombre en donde queramos que se ejecute ese bloque de código.

Recordemos la funcion que declaramos anteriormente.

```
suma(5, 2);
```

El código anterior llama a la función con un argumento de `5` y `2`. La función ejecuta sus declaraciones internas y devuelve el valor `7`.

Las funciones deben llamarse *dentro del ámbito* donde fueron declaradas para que se puedan ejecutar.

El ámbito de una función es la función en la que se declara (o el programa completo, si se declara en el nivel superior).

```
function padre() {
  console.log("padre");
  function hijo() {
    console.log("hijo");
  }
  hijo(); // "hijo"
}

papa(); // "padre"

// que pasa cuando llamo a la funcion hijo que vive dentro de l

hijo(); // => ReferenceError: hijo is not defined
```

Vamos hacer unos ejercicios con todo lo que vimos 😎

1. Hacer una función que reciba un numero y que retorne verdadero si es par o falso en caso contrario.
2. Hacer una función que reciba una serie de palabras separadas por espacios y que imprima la misma frase pero en orden inverso.
3. Hacer una función que reciba como parámetro tres números enteros y que lo ordene de mayor a menor.

Solución 1

```
function par(n) {
  if (n % 2 === 0) {
    return true;
  } else {
    return false;
  }
}

console.log(par(5)); //false
```

Solución 2

```
function invertirFrase(n) {
  let frase = "";
  for (let i = n.length - 1; i >= 0; i--) {
    frase += n[i];
  }
  return frase;
}

console.log(invertirFrase("buenas tardes")); //sedrat saneub
```

Solución 3

```
function minNum(num_1, num_2, num_3) {
  let min = 0;
  if (num_1 > num_2) {
    if (num_3 > num_2) {
      min = num_2;
    } else {
      min = num_3;
    }
  } else {
    if (num_3 > num_1) {
      min = num_1;
    } else {
      min = num_3;
    }
  }
  return min;
}
```

```
function maxNum(num_1, num_2, num_3) {
  let max = 0;
  if (num_1 < num_2) {
    if (num_3 < num_2) {
      max = num_2;
    } else {
      max = num_3;
    }
  } else {
    if (num_3 < num_1) {
      max = num_1;
    } else {
      max = num_3;
    }
  }
  return max;
}
```

```
function ordenar(num_1, num_2, num_3) {
  let max = maxNum(num_1, num_2, num_3);
  let min = minNum(num_1, num_2, num_3);

  let medio = num_1 + num_2 + num_3 - max - min;

  return max + " " + medio + " " + min;
}

console.log(ordenar(33, 77, 36)); // 77 36 33
```