



Técnico en
< DESARROLLO DE SOFTWARE >

Lógica de la Programación II

(CC BY-NC-ND 4.0)
International

Attribution-NonCommercial-NoDerivatives 4.0



Atribución

Usted debe reconocer el crédito de una obra de manera adecuada, proporcionar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que tiene el apoyo del licenciante o lo recibe por el uso que hace.



No Comercial

Usted no puede hacer uso del material con fines comerciales.



Sin obra derivada

Si usted mezcla, transforma o crea un nuevo material a partir de esta obra, no puede distribuir el material modificado.

No hay restricciones adicionales - Usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia.

<http://creativecommons.org/licenses/by-nc-nd/4.0/>



Lógica de la Programación II

Unidad I

Código Modular

Podemos calcular el área de un rectángulo con el siguiente código:

```
var ancho = prompt("Ingrese el ancho: ");  
var alto = prompt("Ingrese el alto: ");  
alert("Área " + ancho*alto);
```

Imagine que se le pide que calcule el área de tres rectángulos diferentes y el código quedaría así:

```
//Primer rectángulo  
var ancho = prompt("Ingrese el ancho: ");  
var alto = prompt("Ingrese el alto: ");  
alert("Área " + ancho*alto);  
  
//Segundo rectángulo  
var ancho = prompt("Ingrese el ancho: ");  
var alto = prompt("Ingrese el alto: ");  
alert("Área " + ancho*alto);  
  
//Tercer rectángulo  
var ancho = prompt("Ingrese el ancho: ");  
var alto = prompt("Ingrese el alto: ");  
alert("Área " + ancho*alto);
```

Función

En programación, a menudo se utiliza código para realizar una tarea específica varias veces. En lugar de reescribir el mismo código, podemos agrupar un bloque de código y asociarlo con una tarea, luego podemos reutilizar ese bloque de código cada vez que necesitemos realizar la tarea nuevamente. Se puede lograr esto creando una **función**.

Una **función** es un bloque de código reutilizable que agrupa una secuencia de instrucciones para realizar una tarea específica.

En esta unidad, aprenderá cómo crear y usar funciones, y cómo se pueden usar para crear código más claro y conciso.

1. Declarar una función

Declarar una función es asociar un conjunto de líneas de código a un identificador o nombre. Es necesario declarar una función antes de poder utilizarla.

Las funciones pueden ser declaradas de la siguiente forma:

Pseudocódigo

```
FUNCION nombre_funcion()
```

```
    bloque de instrucciones
```

```
FIN FUNCION nombre_funcion
```

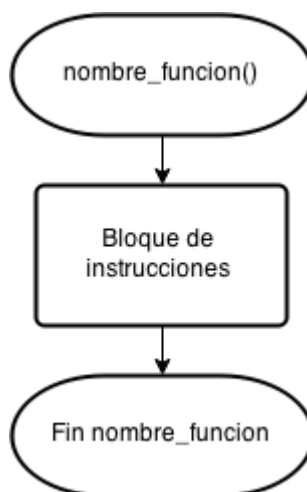
FUNCION: es una palabra reservada. Indica donde comienza el bloque de instrucciones de la función.

nombre_funcion: El nombre de la función está conformado por caracteres alfanuméricos y guiones, no puede iniciar con ningún número o carácter especial. Se colocan los paréntesis porque dentro de ellos se enumeran los parámetros, esto lo veremos más adelante.

FIN FUNCION: es una palabra reservada, indica donde se termina el bloque de instrucciones de la función con el nombre nombre_funcion.

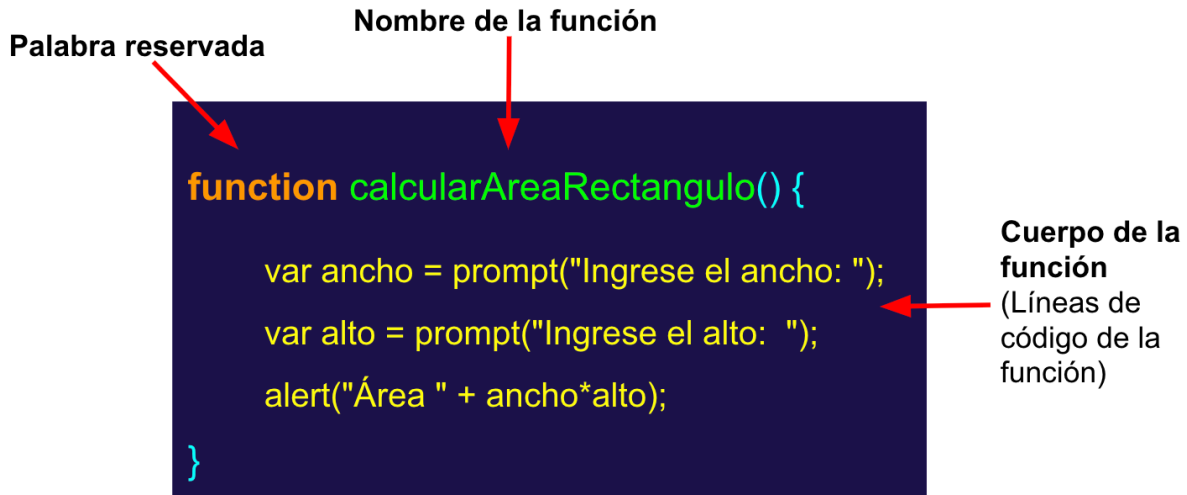
Diagrama de flujo

Por cada función se debe realizar un diagrama de flujo independiente del programa. El diagrama de flujo de una función inicia con el nombre de la función y finaliza con la palabra Fin y el nombre de la función.



Código en JavaScript

En JavaScript, hay varias formas de crear una función. Una forma de crear una función es mediante el uso de una **declaración de función**. Al igual que una declaración de variable vincula un valor a un nombre de variable, una declaración de función vincula una función a un nombre o un identificador. La sintaxis para declarar una función es la siguiente:



Una declaración de función consiste en:

- La palabra reservada **function**.
- El nombre de la función, o su identificador, seguido de paréntesis.
- El cuerpo de una función, o el bloque de instrucciones requeridas para realizar una tarea específica, encerrada en las llaves de la función, {}.

Ejemplo declaración de función:

Crear una función que despliegue en pantalla las vocales, una por una.

Pseudocódigo

FUNCION imprimir_vocales()

imprimir "a"

imprimir "e"

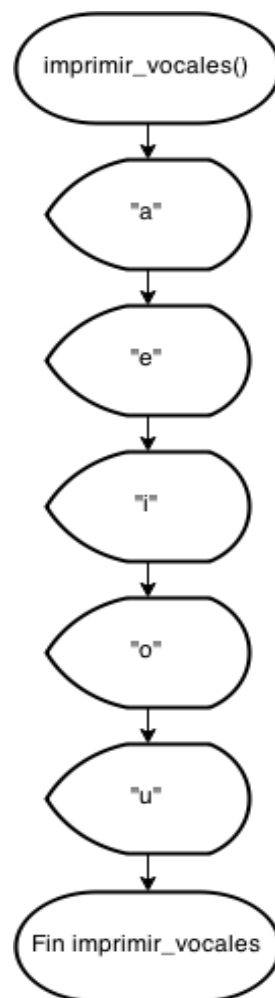
imprimir "i"

imprimir "o"

imprimir "u"

FIN FUNCION imprimir_vocales

Diagrama de Flujo



Código en JavaScript

```
function imprimir_vocales() {  
  
    alert("a");  
  
    alert("e");  
  
    alert("i");  
  
    alert("o");  
  
    alert("u");  
  
}
```

Nota: Si se ejecuta este código no va a pasar nada porque el programa debe mandar a llamar la función. Esto lo veremos en la siguiente sección.

2. Ejecutar una función

Como vimos anteriormente, una declaración de función une un conjunto de líneas de código a un identificador. Sin embargo, una declaración de función no solicita que se ejecute el código dentro del cuerpo de la función, solo declara la existencia de la función. El código dentro del cuerpo de una función se ejecuta o se invoca, solo cuando se llama a la función.

Pseudocódigo

En pseudocódigo se manda a llamar una función con su nombre y sus argumentos dentro de paréntesis. Hasta el momento no hemos visto que son los argumentos pero lo veremos más adelante. Si la función no tiene argumentos se manda a llamar con el nombre de la función, paréntesis abierto y cerrado.

FUNCION *nombre_funcion()*

bloque de instrucciones

FIN FUNCION *nombre_funcion*

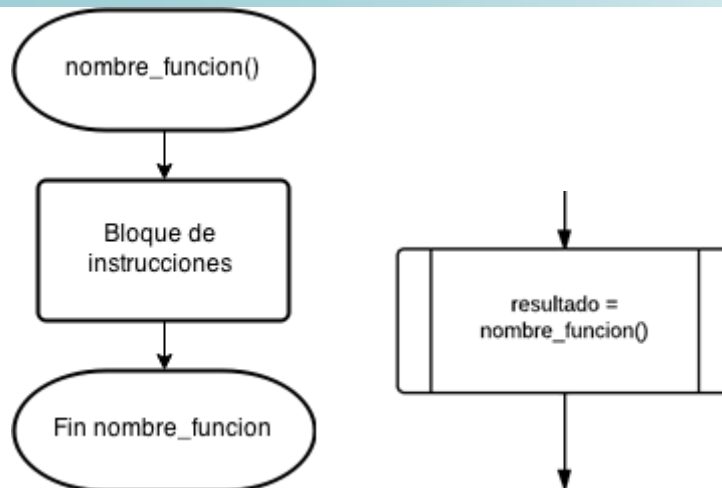
INICIO

nombre_funcion() //Ejecución

FIN

Diagrama de Flujo


Se utiliza el símbolo de subrutina, Es un marcador que representa un segmento de código que está formalmente declarado en otro lugar. Este símbolo se coloca dentro del flujo del programa para mandar a ejecutar la función dentro del bloque de código de acuerdo a las necesidades del programa.



JavaScript

En JavaScript se manda a llamar de forma similar que en pseudocódigo. Se escribe el nombre de la función seguido de paréntesis.

Nombre de la función



```
calcularAreaRectangulo();
```

Esta llamada a la función ejecuta el cuerpo de la función o todas las líneas de código que se encuentran entre las llaves de la declaración de la función.

También debemos tener en cuenta el funcionamiento de elevación o hoisting en JavaScript, esto permite ejecutar las funciones antes de que se definan.

```
saludar();  
function saludar() {  
    alert("Hola");  
}
```

Observe como la elevación permitió que se llamara saludar() antes de que se definiera la función. Dado que la elevación a veces puede dar conflictos en los programas, es importante que conozca este concepto para evitar hacerlo.

Si desea leer más sobre la elevación o hoisting, puede consultar el siguiente [enlace](#).

Ejemplo ejecutar funciones:

Vamos a continuar con el ejemplo anterior de crear un programa que despliegue en pantalla las vocales dos veces, una por una, solo que ahora vamos a mandar a ejecutar la función.

Pseudocódigo

FUNCION imprimir_vocales()

imprimir "a"

imprimir "e"

imprimir "i"

imprimir "o"

imprimir "u"

FIN FUNCION imprimir_vocales

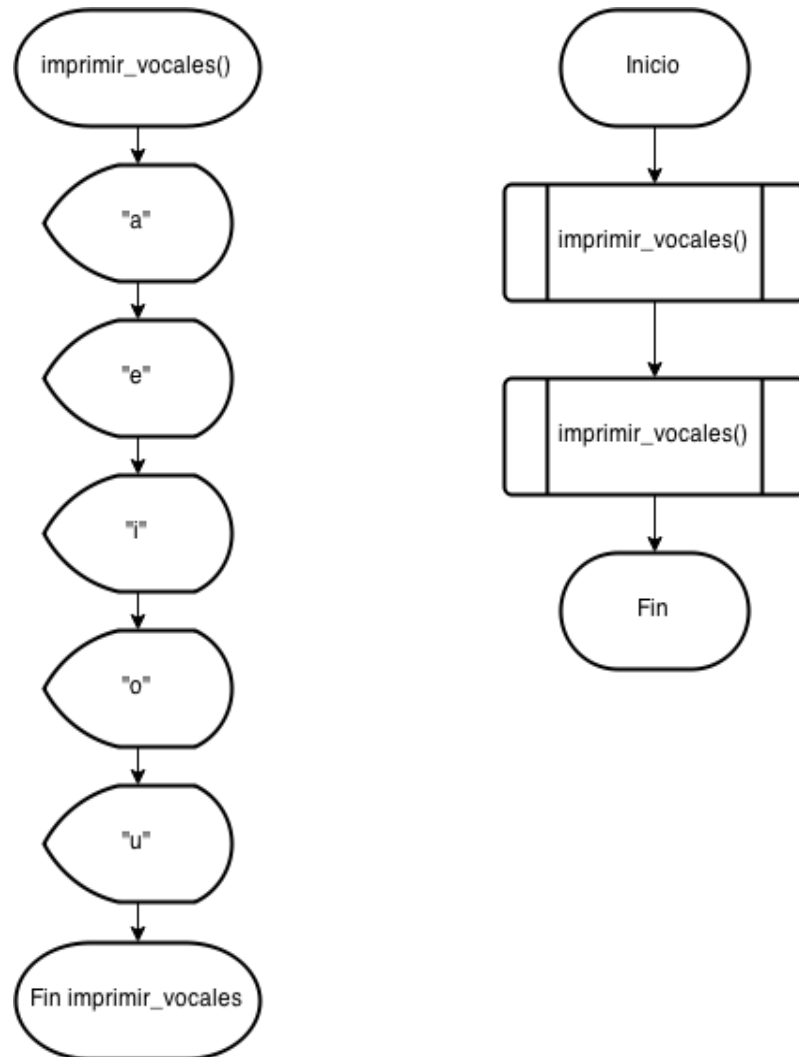
INICIO

imprimir_vocales()

imprimir_vocales()

FIN

Diagrama de Flujo



Código en JavaScript

```
function imprimir_vocales() {  
    alert("a");  
    alert("e");  
    alert("i");  
    alert("o");  
    alert("u");  
}  
imprimir_vocales();  
imprimir_vocales();
```

3. Parámetros y argumentos

Hasta ahora, las funciones que hemos creado ejecutan una función sin una entrada. Sin embargo, algunas funciones pueden tomar entradas y usar las entradas para hacer cálculos o tareas específicas dentro de la función.

Estas entradas se denominan parámetros, al declarar una función se pueden especificar sus parámetros dentro de los paréntesis separados por coma. Los parámetros se utilizan como contenedores para la información que se pasará a la función cuando esta se ejecuta. Observemos cómo especificar parámetros en nuestra declaración de función en JavaScript:

Parámetros

```
function calcularAreaRectangulo(ancho, alto) {  
    alert("Área " + ancho*alto);  
}
```

Los parámetros son tratados
como variables en la función

La función `calcularAreaRectangulo` calcula el área de un rectángulo en base a dos entradas: `ancho` y `alto`. Los parámetros se especifican entre paréntesis como `ancho` y `alto`, y actúan como variables regulares dentro del cuerpo de la función.

Al ejecutar o mandar a llamar una función que tiene parámetros, especificamos los valores entre paréntesis que siguen al nombre de la función. Los valores que se pasan a la función cuando se ejecuta se denominan argumentos. Los argumentos se pueden pasar a la función como valores o variables.

Valores como
argumentos

```
calcularAreaRectangulo(8, 3);
```

Nombre de la función

En la llamada a la función anterior, el número 8 se pasa como el ancho y el número 3 se pasa como el alto. Observe que el orden en que se pasan y asignan los argumentos sigue el orden en que se declaran los parámetros.

También se pueden pasar variables al mandar a ejecutar una función:

```
var anchoRectangulo = 8;  
var altoRectangulo = 3;  
calcularAreaRectangulo(anchoRectangulo, altoRectangulo);
```

Nombre de la función

Valores como argumentos

Las variables anchoRectangulo y altoRectangulo se inicializan con los valores del ancho y alto de un rectángulo antes de usarse en la llamada a la función.

La función calcularAreaRectangulo al usar parámetros se puede reutilizar para calcular el área de cualquier rectángulo. Las funciones son una herramienta poderosa en la programación de computadoras.

Ejemplo parámetros y argumentos:

Crear una función que reciba dos valores enteros e imprima en pantalla el resultado de la multiplicación de los dos valores.

Pseudocódigo

Esta función tendrá dos parámetros que son los números a multiplicar. Para poder

obtener el resultado se mandará a ejecutar la función con los números 6 y 4 como argumentos.

FUNCION multiplica(multiplicando1:entero,multiplicando2:entero)

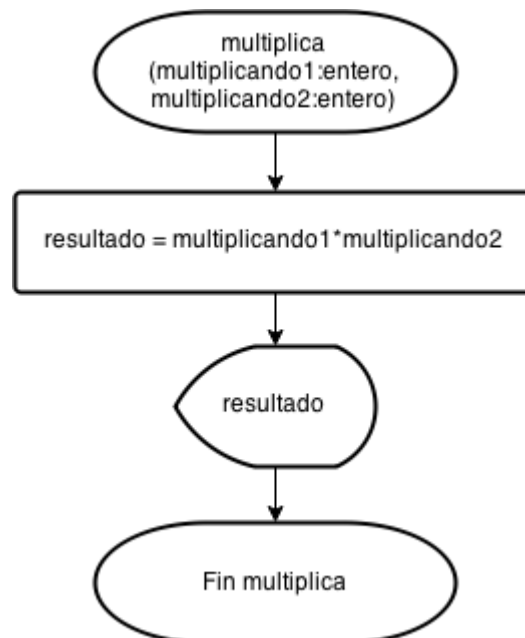
definir resultado \leftarrow multiplicando1 * multiplicando2

imprimir resultado

FIN FUNCION multiplica

multiplica(6,4)

Diagrama de Flujo



Código en JavaScript

```
function multiplica(multiplicando1,multiplicando2){  
  
    //multiplicando1 y multiplicando2 son parámetros de multiplica  
  
    var resultado = multiplicando1 * multiplicando2;  
  
    // Imprime el resultado de la multiplicación  
  
    imprimir resultado;  
  
}  
  
multiplica (6,4); // Se llama la función con los argumentos 6 y 4
```

Evolución de JavaScript: ES5 a ES6

Con cada nueva actualización de un programa, el objetivo es mejorar siempre la calidad y la funcionalidad. El primer cambio importante para JavaScript ocurrió en 2009, donde se introdujo ECMAScript 5 (ES5) y nacieron muchas características excelentes. Se introdujeron funciones como reducir y filtrar en este parche. El equipo de JavaScript no se detuvo al intentar crear JavaScript como un lenguaje más estándar. Seis años más tarde, ECMAScript 6 (ES6) se convirtió en el nuevo estándar de JavaScript para implementar. La actualización trató de hacer que la escritura de JavaScript fuera menos confusa y las herramientas que se introdujeron, fueron diseñadas para que los desarrolladores se comuniquen mejor entre sí. Hay muchos cambios que cubrir, pero veremos en esta unidad algunas funcionalidades recientes relacionadas a las funciones.

Parámetros por defecto

Una de las características agregadas en ES6 es la capacidad de usar parámetros predeterminados. Los parámetros predeterminados permiten que los parámetros tengan un valor predeterminado en caso de que no se pase ningún argumento a la función o si el argumento no está definido cuándo se ejecuta la función.

Por ejemplo:

```
function alquilarCarro (anio = 2020, color = 'azul') {  
    alert('Alquilar carro - Color: ' + color + ' Año: ' + anio);  
}  
alquilarCarro(2018,'verde');  
alquilarCarro();
```

Se utiliza el operador = para asignar al nombre del parámetro un valor predeterminado, en el ejemplo se colocó el año 2020 y el color azul como valores. Es útil cuando no especifican el carro que desean alquilar y se les proporciona el carro predeterminado.

Cuando en el código se llama a *alquilarCarro(2018,'verde')*; se pasan los valores de los argumentos. El valor 2018 anulará el parámetro predeterminado de 2020 y el valor 'verde' anulará el parámetro predeterminado de 'azul'. Dando como resultado: *'Alquilar carro – Color: verde Año: 2018'*.

Al usar un parámetro predeterminado, tenemos en cuenta las situaciones en las que un argumento no se pasa a una función que espera un argumento.

4. Valores de Retorno

Cuando se llama a una función, la computadora ejecutará el código de la función y evaluará el resultado de llamar a la función. Por defecto, ese valor resultante no está definido.

```
function calcularAreaRectangulo(ancho, alto) {  
    var area = ancho*alto;  
}  
alert(calcularAreaRectangulo(4, 6));
```

En el ejemplo de código, se definió la función para calcular el área de un rectángulo con los parámetros de ancho y alto. Luego se invoca `calcularAreaRectangulo` con los argumentos 4 y 6. Pero cuando se mostró el resultado, mostro *undefined* (indefinido). ¿Escribimos mal nuestra función? No, de hecho la función funcionó bien y la computadora calculó el área como 24, pero no se devolvió nada al ejecutar la función.

Para devolver información al ejecutar una función, utilizamos una declaración de devolución. Para crear una declaración de devolución, utilizamos la palabra reservada **return** seguida del valor que deseamos devolver. Como vimos anteriormente, si se omite el `return` dentro del código de la función se devuelve un valor indefinido (*undefined*) en su lugar.

```
function calcularAreaRectangulo(ancho, alto) {  
    var area = ancho*alto;  
    return area;  
}  
alert(calcularAreaRectangulo(4, 6));
```

Cuando se usa una declaración **return** en el cuerpo de una función, la ejecución de la función se detiene y el código que sigue no se ejecutará. Por ejemplo:

```
function calcularAreaRectangulo(ancho, alto) {  
    if (ancho < 0 || alto < 0) {  
        return 'Ancho y alto deben ser enteros positivos';  
    }  
    return ancho*alto;  
}  
alert(calcularAreaRectangulo(4, 6));
```

Si un argumento para ancho o alto es menor que 0, calcularAreaRectangulo devolverá *'Ancho y alto deben ser enteros positivos'*. La segunda declaración de retorno ancho * alto no se ejecutará.

La palabra clave **return** es poderosa porque permite que las funciones produzcan una salida. Luego podemos guardar la salida en una variable para su uso posterior.

Ejemplo valores de retorno:

Hacer un programa que sume los números 4 y 3 utilizando código modular:

Pseudocódigo

En pseudocódigo se utiliza la palabra reservada "devolver", para indicar el valor o la variable que será retornada al finalizar la función.

FUNCION suma(num1:entero,num2:entero) **DEVOLVER** (resultado:entero)

resultado ← num1 + num2

FIN FUNCION suma

INICIO

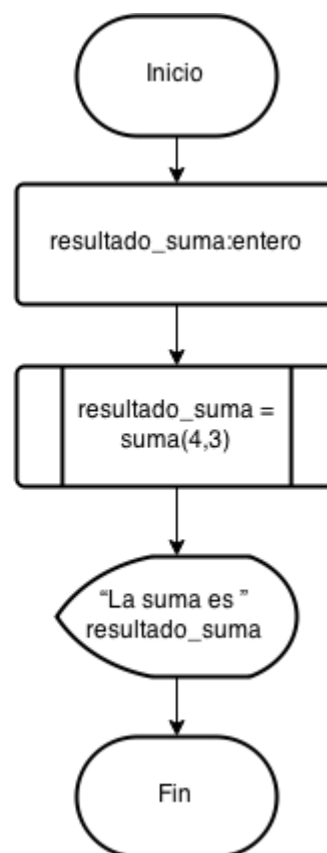
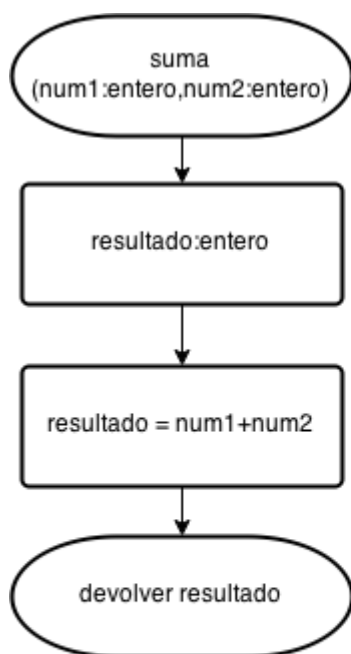
definir resultado_suma:entero

resultado_suma \leftarrow suma(4,3)

imprimir "La suma es " resultado_suma

FIN

Diagrama de Flujo



Código en Javascript

```
function suma(num1,num2){  
    var resultado = num1 + num2;// Se realiza la suma de los parámetros  
    return resultado; // Se devuelve el valor almacenado en resultado  
}  
  
/*Se realiza el llamado a la función y el valor retornado se almacena en la variable  
resultado_suma. En este caso resultado_suma contendría el valor de 7. */  
  
var resultado_suma = suma(4,3);  
  
/* Muestra una ventana con el texto La suma es 7*/  
  
alert("La suma es: " + resultado_suma);
```

Ejemplo valores de retorno:

Hacer un programa utilizando funciones que solicite al usuario el ingreso de un número y muestre en pantalla el resultado del cuadrado de un número.

Pseudocódigo

FUNCION cuadrado(numero:entero) **DEVOLVER** (resultado:entero)

resultado ← numero * numero

FIN FUNCION cuadrado

INICIO

definir numero:entero

definir total:entero

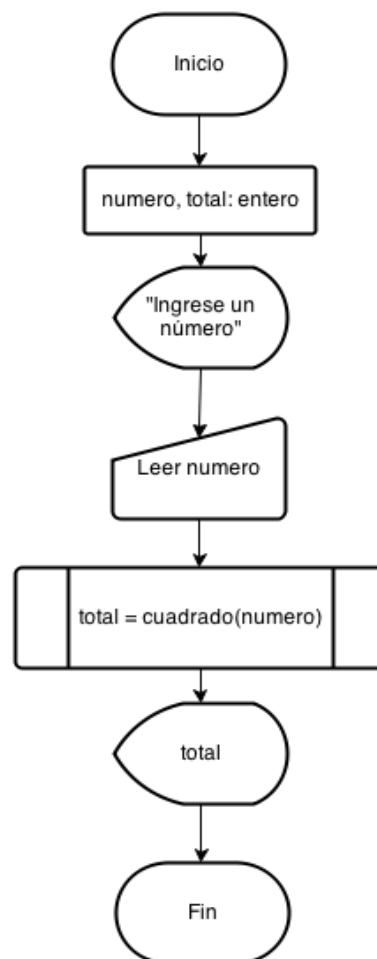
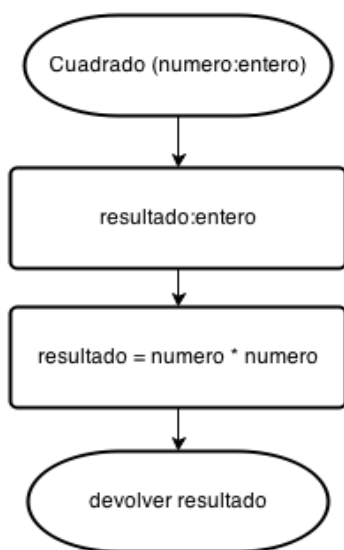
numero \leftarrow leer numero

total \leftarrow cuadrado(numero)

imprimir total

FIN

Diagrama de flujo



Código en Javascript

```
function cuadrado(numero){  
    var resultado = numero * numero;  
    return resultado;  
}  
  
var numero, total;  
  
numero = parseInt(prompt("Ingrese un numero");  
  
total = cuadrado(numero);  
  
alert(total);
```

Funciones auxiliares

Se puede mandar a ejecutar una función dentro de otra función. Estas funciones que se llaman dentro de otra función a menudo se denominan funciones auxiliares. Dado que cada función está llevando a cabo una tarea específica, hace que nuestro código sea más fácil de leer y depurar si es necesario.

Si quisiéramos definir una función que obtenga el precio de venta con IVA y aplicando un descuento, podríamos escribir dos funciones como:





```
function calcularDescuento(precio,descuento) {  
    return precio - precio*descuento;  
}  
function precioFinal(precio,descuento) {  
    precio = calcularDescuento(precio,descuento);  
    return precio + precio*0.12;  
}  
  
alert(precioFinal(82.00, 0.05));
```

Podemos usar funciones para seccionar pequeños fragmentos de lógica o tareas, y luego usarlas cuando sea necesario. Escribir funciones auxiliares puede ayudar a realizar tareas grandes y difíciles y dividir las en tareas más pequeñas y manejables.

Expresión de función

Es una variación para definir funciones como parte de una expresión. En una expresión de función, el nombre de la función generalmente se omite. Una función sin nombre se llama **función anónima**. Sin embargo, debido a que una función es un tipo de datos, puede asignarse a una variable. La variable se utiliza para invocar la función más adelante.

Considere la siguiente expresión de función:

Identificador	Palabra reservada	Parámetros
		
<pre>const calcularAreaRectangulo = function (ancho, alto) { const area = ancho*alto; return area; }</pre>		

Para declarar una expresión de función:

- Declare una variable para que el nombre de la variable sea el nombre o el identificador de su función. Desde el lanzamiento de ES6, es una práctica común usar `const` como palabra clave para almacenar en una constante la función.
- Asigne como valor de esa variable una función anónima creada utilizando la palabra reservada **function** seguida de un conjunto de paréntesis con posibles parámetros. Luego, un conjunto de llaves que contienen el cuerpo de la función.

Para invocar una expresión de función se hace igual que con las funciones definidas, se escribe el nombre de la variable en la que se almacena la función seguido de paréntesis que encierran los argumentos que se pasan a la función.

`nombreFuncion(argumento1,argumento2);`

A diferencia de las declaraciones de función, las expresiones de función no se elevan, por lo que no se pueden invocar antes de definirse.

En resumen, use declaraciones de funciones cuando desee crear una función en el alcance global y ponerla a disposición en todo su código. Use expresiones de función para limitar dónde está disponible la función, mantener su alcance global ligero y mantener una sintaxis limpia.

Funciones de flecha

Las funciones de flecha o Arrow Functions son una nueva sintaxis para definir funciones anónimas en JavaScript de un modo más conciso. ES6 introdujo la sintaxis de la función de flecha, una forma más corta de escribir funciones mediante el uso de la notación especial `() =>`. Las funciones de flecha eliminan la necesidad de escribir la palabra reservada *function* cada vez que necesita crear una función. En su lugar, primero incluye los parámetros dentro de paréntesis `()` y luego agrega una flecha `=>` que apunta al cuerpo de la función rodeado de llaves `{}` así:

```
const calcularAreaRectangulo = (ancho, alto) => {  
  var area = ancho*alto;  
  return area;  
}
```

Es importante estar familiarizado con las múltiples formas de escribir funciones porque se encontrará con cada una de ellas cuando lea otro código JavaScript.

Funciones de flecha concisas

JavaScript también proporciona varias formas de refactorizar la sintaxis de la función de flecha. La forma más condensada de la función se conoce como cuerpo conciso. Exploraremos algunas de estas técnicas a continuación:

1. Las funciones que toman un solo parámetro no necesitan que ese parámetro esté encerrado entre paréntesis. Sin embargo, si una función toma cero o múltiples parámetros, se requieren paréntesis.
2. Un cuerpo de función compuesto por un bloque de una sola línea no necesita

llaves. Sin las llaves, lo que evalúa esa línea se devolverá automáticamente. El contenido del bloque debe seguir inmediatamente a la flecha => y se puede eliminar la palabra clave return. Esto se conoce como retorno implícito.

Entonces si tenemos una función:

```
const elevarAlCuadrado = (num) => {  
  return num*num;  
}
```

Podemos refactorizar la función así:

```
const elevarAlCuadrado = num => num*num;
```

Observe los siguientes cambios:

- Los paréntesis alrededor de num se han eliminado, ya que tiene un solo parámetro.
- Las llaves {} se han eliminado ya que la función consiste en un bloque de una sola línea.
- La palabra clave return se ha eliminado ya que la función consiste en un bloque de una sola línea.

4. Alcance

Una idea importante en la programación es el alcance o scope en inglés. El alcance define dónde se puede acceder o hacer referencia a las variables. Si bien se puede acceder a algunas variables desde cualquier lugar dentro de un programa, otras variables solo pueden estar disponibles en un contexto específico.

Puede pensar en el alcance como la vista del cielo nocturno desde su ventana. Todos los que viven en el planeta Tierra están en el alcance global de las estrellas. Las estrellas son accesibles a nivel mundial. Mientras tanto, si vives en una ciudad, puedes ver el horizonte de la ciudad o el río. El horizonte y el río solo son accesibles localmente en su ciudad, pero aún puede ver las estrellas que están disponibles en todo el mundo.

Bloques y alcance de las variables

Antes de hablar más sobre el alcance, primero tenemos que tener claro el concepto de los bloques.

Un bloque es el código que se encuentra dentro de un conjunto de llaves {}. Los bloques nos ayudan a agrupar una o más declaraciones y sirven como un marcador estructural importante para nuestro código. Hemos visto bloques usados antes en funciones y en declaraciones de if. Un bloque de código podría ser una función como esta:

```
function calcularPrecioTotal (precio) {  
    var iva = 0.12;  
    return precio + precio*iva;  
}
```

Observe que el cuerpo de la función es en realidad un bloque de código.

Observe el bloque en una declaración if:

```
if (descuento) {  
    var porcentaje = '12%';  
    console.log(porcentaje);  
}
```

Alcance global

El alcance es el contexto en el que se declaran nuestras variables. Pensamos en el alcance en relación con los bloques porque las variables pueden existir fuera o dentro de estos bloques.

En el ámbito global, las variables se declaran fuera de los bloques. Estas variables se llaman variables globales. Debido a que las variables globales no están vinculadas dentro de un bloque, se puede acceder a ellas mediante cualquier código del programa, incluido el código en bloques.

Veamos un ejemplo en JavaScript:

```
var color = azul;  
function colorDelCielo () {  
    return color;  
}  
alert(colorDelCielo());
```

- Aunque la variable color se define fuera del bloque, se puede acceder a ella en el bloque de la función, lo que le otorga un alcance global.
- A su vez, se puede acceder al color dentro del bloque de la función colorDelCielo.

Aquí pueden ver otro ejemplo en pseudocódigo:

```
definir medioTransporte:caracter

medioTransporte ← "Avión";

// medioTransporte puede usarse aquí

FUNCION transporteActual() DEVOLVER (medioTransporte:caracter)

    //medioTransporte puede usarse en este bloque de código

FIN FUNCION transporteActual

mostrar(transporteActual());
```

Alcance local o de bloque

El siguiente contexto es el alcance del bloque. Cuando una variable se define dentro de un bloque, solo es accesible para el código dentro de las llaves {}. Decimos que la variable tiene alcance de bloque porque solo es accesible a las líneas de código dentro de ese bloque.

Las variables que se declaran con alcance de bloque se conocen como variables locales porque solo están disponibles para el código que forma parte del mismo bloque.

Veamos un ejemplo en JavaScript:

```
function colorDelCielo () {  
    var color = 'azul';  
    alert(color);  
}  
alert(colorDelCielo());  
alert(color); //Error de referencia
```

- Definimos una función llamada colorDelCielo().
- Dentro de la función, la variable color solo está disponible dentro de las llaves de la función.
- Si intentamos utilizar la variable color fuera de la función, muestra el error:
"ReferenceError: color is not defined."

Es importante tomar en cuenta el tiempo de vida de las variables en JavaScript. La vida de una variable empieza cuando es declarada. En el caso de variables locales, son destruidas cuando termina la ejecución de la función. Las variables globales son destruidas cuando se cierra la página.

Aquí pueden ver otro ejemplo en pseudocódigo:

FUNCION carroActual()

/* Al declararse la variable nombreCarro puede ser utilizada en este bloque de código */

definir nombreCarro:caracter

nombreCarro ← "Sedan"


```
imprimir nombreCarro
```

```
FIN FUNCION carroActual
```

```
INICIO
```

```
/* Dentro del programa no se puede utilizar la variable nombreCarro aunque se  
mande a ejecutar la función */
```

```
carroActual();
```

```
FIN
```

Contaminación del alcance

Puede parecer una gran idea hacer siempre accesibles sus variables, pero tener demasiadas variables globales puede causar problemas en un programa.

Cuando declara variables globales, van al espacio de nombres global. El espacio de nombres global permite que las variables sean accesibles desde cualquier parte del programa. Estas variables permanecen allí hasta que finaliza el programa, lo que significa que nuestro espacio de nombres global puede llenarse muy rápidamente.

La contaminación del alcance es cuando tenemos demasiadas variables globales que existen en el espacio de nombres global, o cuando reutilizamos variables en diferentes ámbitos. La contaminación del alcance dificulta el seguimiento de nuestras diferentes

variables y nos prepara para posibles accidentes. Por ejemplo, las variables de alcance global pueden colisionar con otras variables que tienen un alcance más local, causando un comportamiento inesperado en nuestro código.

Por ejemplo:

```
var color = 'verde';

function mostrarColor() {
    color = 'rojo';
    alert(color);
}
mostrarColor(); //Muestra rojo
alert(color); //Muestra rojo
```

- Tenemos una variable color.
- Cuando llamamos a mostrarColor(), se cambia el color a 'rojo'.
- La reasignación dentro de mostrarColor() afecta la variable global color.
- Aunque la reasignación está permitida y no obtendremos un error, si decidimos usar color más tarde, sin saberlo, tendremos el color 'rojo' en lugar del color esperado 'verde'.

Si bien es importante saber que es el alcance global, es una buena práctica no definir variables en el alcance global.

Dados los desafíos con las variables globales y la contaminación del alcance, debemos seguir las mejores prácticas para determinar nuestras variables lo más estrictamente posible usando el alcance de bloque.

Al definir con precisión sus variables, su código mejorará de varias maneras:

- Hará que su código sea más legible ya que los bloques organizarán su código en secciones discretas.
- Hace que su código sea más comprensible ya que aclara qué variables están asociadas con diferentes partes del programa en lugar de tener que realizar un seguimiento línea por línea.
- Es más fácil mantener su código, ya que su código será modular.
- Ahorrará memoria en su código porque dejará de existir una vez que el bloque termine de ejecutarse.

5. Recursión

Es un tipo de función que se manda a llamar a sí misma para obtener un resultado final. Para este tipo de funciones, es necesario definir bien las condiciones para evitar que la recursión sea infinita. Este tipo de funciones es muy útil cuando se necesitan calcular resultados complejos como el caso de encontrar un factorial. Para encontrar el factorial de un número se debe multiplicar el número inicial por el número siguiente menos uno hasta llegar a uno.

Veamos un ejemplo para entenderlo mejor:

```
function factorial(x) {  
  if (x < 0) return;  
  if (x === 0) return 1;  
  return x * factorial(x - 1);  
}  
factorial(4); // 24
```

Por ejemplo, para encontrar el factorial de 6 sería:

$$6 * 5 * 4 * 3 * 2 * 1 = 720$$

Como puede ver, realizar este cálculo sería mucho más complejo de manera iterativa. Otro uso común del uso de funciones recursivas es en la navegación de árboles binarios en donde se desconoce el tamaño de las ramas y la cantidad de nodos que posea el mismo.

Existen tres elementos claves que deben especificarse al definir funciones recursivas. Estos elementos son los siguientes:

1. Caso base:

Es una condición que debe definirse para que la ejecución termine. El caso base de la función factorial es el factorial de cero:

$$\text{factorial}(0) = 1$$

2. Condición de finalización:

Es necesario delimitar los casos que no sean definidos para las funciones recursivas para evitar el bloqueo del programa en tiempo de ejecución. En el caso de la función factorial se conoce que no está definida esta función para números negativos, por lo cual debemos colocar una condición como la siguiente:

```
if(x < 0 ) return;
```

3. La recursión como tal:

Para poder hacer uso de los beneficios de una función recursiva se debe incluir la invocación a esta misma o de lo contrario la función se ejecutaría únicamente una vez. Retornando al ejemplo, la recursión es la siguiente:

```
return x * factorial(x-1);
```

A continuación, se muestran todos los elementos en conjunto:

```
function factorial(x) {  
  
    // Condición de finalización  
    if (x < 0) return;  
  
    // Caso base  
    if (x === 0) return 1;  
  
    // Recursión  
    return x * factorial(x - 1);  
  
}  
  
factorial(5); // 120
```

Para entenderlo de una mejor manera a continuación se presenta el flujo de ejecución y retorno de resultados para encontrar el factorial del número tres.

Flujo de ejecución:

```
factorial(3) //devuelve 3 * factorial(2)
      ↓
factorial(2) //devuelve 2 * factorial(1)
      ↓
factorial(1) //devuelve 1 * factorial(0)
      ↓
factorial(0) //devuelve 1
```

Cuando se alcanza el caso base, la función empieza a retornar en orden inverso ya que los valores de retorno ya se encuentran definidos y puede continuar realizando los cálculos requeridos:

```
factorial(0) //devuelve 1                => 1
      ↓
factorial(1) //devuelve 1 * factorial(0)  => 1 * 1
      ↓
factorial(2) //devuelve 2 * factorial(1)  => 2 * 1 * 1
      ↓
factorial(3) //devuelve 3 * factorial(2)  => 3 * 2 * 1 * 1
// 3 * 2 * 1 * 1 = 6
```

Como ejemplo adicional podemos definir una función recursiva que devuelva una cadena de caracteres en el orden inverso al cual recibe de argumento. Para esto se utiliza la función de JavaScript `substr()` que devuelve el valor de una cadena de caracteres

truncada a partir del valor que recibe como parámetro. La definición de la función es la siguiente:

```
reversa(cadena){  
    if(cadena === '') return '';  
    return reversa(cadena.substr(1)) + cadena[0];  
}  
  
reversa('pez');  
//zep
```

Identifiquemos los elementos fundamentales:

- **Caso base**
El caso base para este ejemplo es `cadena === ''`
- **Condición de finalización**
En este caso es la misma condición del caso base
- **Recursión**
La recursión de esta función es: `return reversa(cadena.substr(1)) + cadena[0];`

El flujo de ejecución es el siguiente:

```
reversa('pez') //devuelve  reversa('ez') + 'p';  
    ↓  
reversa('ez') //devuelve  reversa('z') + 'e';  
    ↓  
reversa('z') //devuelve  reversa('') + 'z';  
    ↓  
reversa('') //devuelve '';
```

En cuanto se llega al caso base, se empieza el retorno de los valores y esto ocurre de la siguiente manera:

```
reversa('') // devuelve '' => '';  
    ↓  
reversa('z') // devuelve 'z' => '' + 'z';  
    ↓  
reversa('ez') // devuelve 'e' => '' + 'z' + 'e';  
    ↓  
reversa('pez') // devuelve 'p' => '' + 'z' + 'e' + 'p';  
//Resultado: zep
```

A continuación, pueden ver otro ejemplo:

Hacer un programa que mediante funciones que calcule la potencia de un número, el programa debe permitir el ingreso de la base y el exponente.

Pseudocódigo

FUNCION potencia(base:entero,exponente:entero)

SI exponente == 0 **ENTONCES**

DEVOLVER 1

SINO

definir resultado \leftarrow base * potencia(base, exponente - 1)

DEVOLVER resultado

FIN SI

FIN FUNCION potencia

INICIO

definir base:entero

definir exponente:entero

definir resultado:entero

imprimir "Ingrese la base"

base \leftarrow leer numero

imprimir "Ingrese el exponente"

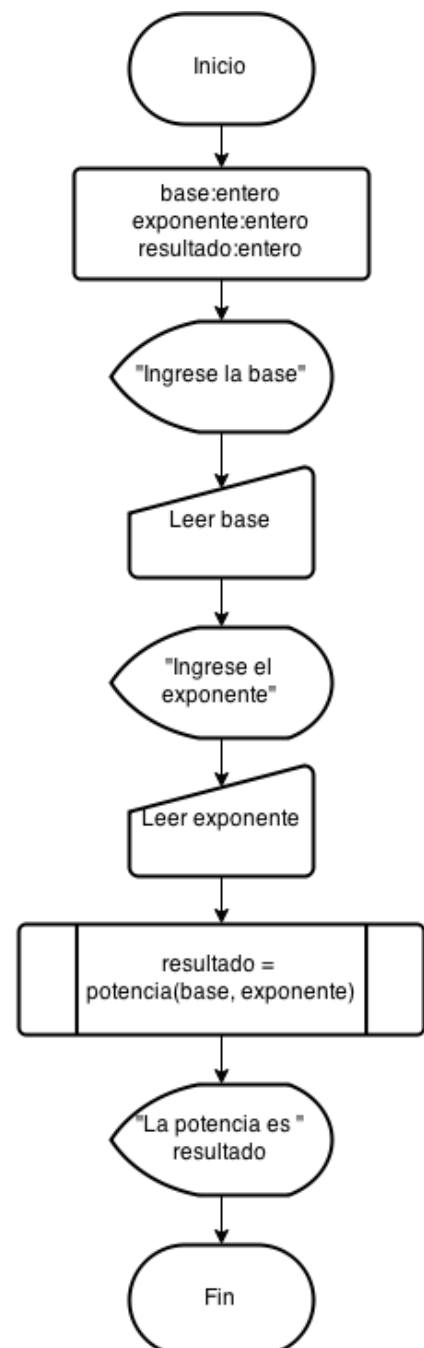
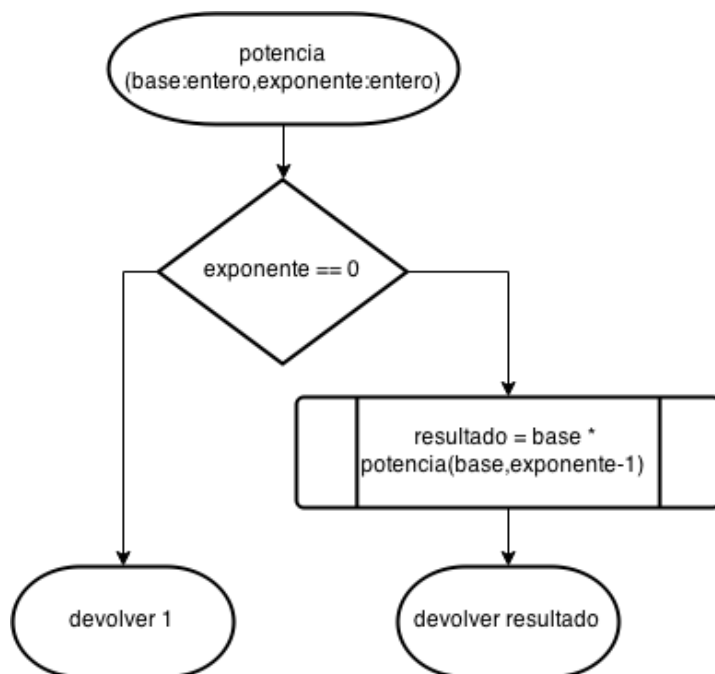
exponente \leftarrow leer numero

resultado \leftarrow potencia(base,exponente)

imprimir "La potencia es " resultado

FIN

Diagrama de Flujo



Código en JavaScript

```
function potencia(base, exponente) {  
  if (exponente == 0)  
    return 1;  
  else  
  
    resultado = base * potencia(base, exponente - 1);  
    return resultado  
}  
  
var base = parseInt(prompt("Ingrese la base"));  
  
var exponente = parseInt(prompt("Ingrese el exponente"));  
  
var resultado = potencia(base,exponente);  
  
alert("La potencia es " + resultado);
```

Descargo de responsabilidad

La información contenida en este documento descargable en formato PDF o PPT es un reflejo del material virtual presentado en la versión online del curso. Por lo tanto, su contenido, gráficos, links de consulta, acotaciones y comentarios son responsabilidad exclusiva de su(s) respectivo(s) autor(es) por lo que su contenido no compromete al área de e-Learning del Departamento GES o al programa académico al que pertenece.

El área de e-Learning no asume ninguna responsabilidad por la actualidad, exactitud, obligaciones de derechos de autor, integridad o calidad de los contenidos proporcionados y se aclara que la utilización de este descargable se encuentra limitada de manera expresa para los propósitos educativos del curso.



Galileo
UNIVERSIDAD
La Revolución en la Educación

GES
Galileo Educational System