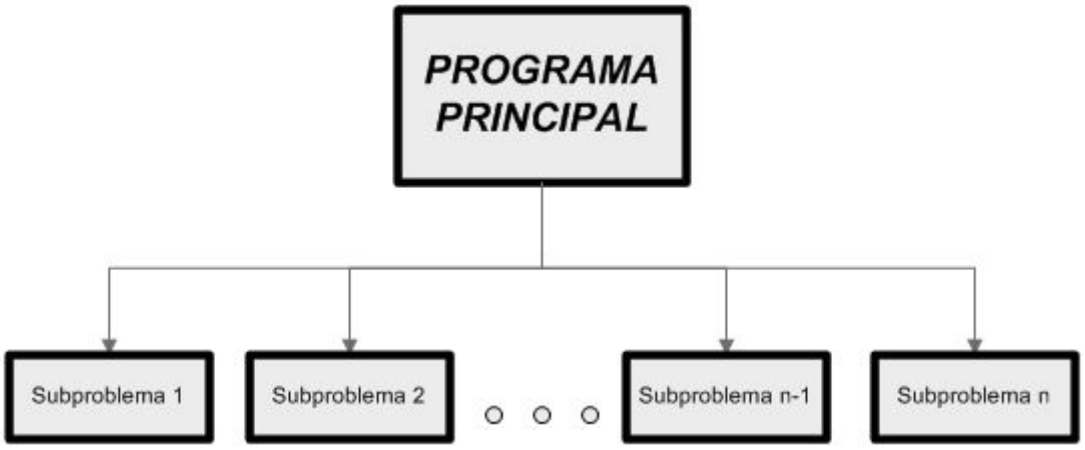
MODULARIZACIÓN Y FUNCIONES
CREACIÓN DE RUTINAS / MÓDULOS



¿Qué es un módulo (o rutina)?

Un módulo es una “pieza” la cual forma parte del programa principal. Cada módulo se encarga de ejecutar una funcionalidad específica: resuelve X problema en particular. Todo módulo tiene un nombre identificador asociado, el cual permitirá ejecutarlo (“invocar” / “llamar” al módulo). Ejemplos: calcularDeuda – borrarDatos – imprimirReporte - etc.

¿Por qué crear módulos?

Crear un programa completo sin la utilización de módulos específicos, conlleva a crear 1 único proceso extenso que resuelva todos requerimientos de la aplicación; lo cual puede ser complejo de comprender y mantener en el tiempo: A medida que crece la aplicación, se vuelve más difícil encontrar en qué lugar específico del código fuente se resuelve X situación en particular.

Mediante la creación de módulos se divide (se descompone) un problema complejo y extenso en pequeños subproblemas, delegando así funcionalidades específicas y acotadas a distintas piezas del programa, las cuales tendrán cada una su responsabilidad.

La modularización es considerada una muy buena práctica de programación: Ya no estaremos programando 1 aplicación de 1 única pieza, estaremos programando muchos “subprogramas”, los cuales entre todos hacen al programa principal.

Beneficios de la modularización:

A. **Delegación de funcionalidades / tareas:**

   Se podrán crear distintos módulos auxiliares, cada uno encargado de resolver X tarea específica.

B. **Mantenibilidad:**

   Siempre será más sencillo modificar una pequeña pieza específica del programa, que modificar un programa que no modulariza: es decir, un programa de una única pieza.

C. **Re-usabilidad / Re-utilización:**

   Una gran ventaja de los módulos es que éstos son re-utilizables.

   Un módulo “A” puede ser ejecutado una y otra vez, por el programa principal, o bien ser ejecutado por un módulo “B”, el cual requiera la funcionalidad del módulo “A”.

   Con la creación de módulos se evita en gran medida la mala práctica del copy-paste de código.

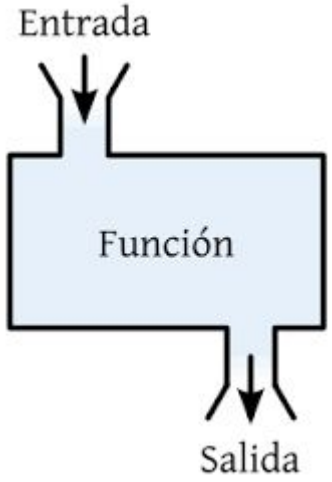
Tipos de módulos

En distintos lenguajes de programación nos encontramos con módulos de tipo:

- Función
- Procedimiento

En algunos lenguajes dispondremos solo de funciones, en otros también existirán los procedimientos.

¿Qué es una función?



Las funciones son módulos capaces de retornar un valor final, como resultado de su proceso. Estos módulos tendrán lo que se conoce como “valor de salida” o también llamado “valor de retorno”.

Ejemplos:

- Una función que se encargue de calcular la edad de una persona retornará dicha edad (valor numérico).

- Una función que se encargue de averiguar si una persona es soltera o no, retornará un valor booleano.

El tipo de dato del valor de retorno puede ser cualquiera de los existentes en el lenguaje de programación en uso. En algunos lenguajes, el tipo de dato del valor de retorno debe ser declarado, en otros lenguajes NO se declara.
En Javascript las funciones son un conjunto de sentencias con un nombre de referencia sin necesariamente tener un valor de retorno. Crear funciones resulta una buena práctica de programación, ya que permite la reusabilidad de recursos. Creando funciones, evitamos en gran medida el copy – paste de código reiteradas veces.

¿Qué es un procedimiento?

Los procedimientos son similares a las funciones, ya que pueden procesar un conjunto de instrucciones que resuelvan X problema en particular; la diferencia con las funciones es que los procedimientos NO devuelven un valor final: Se tratan de módulos que NO tienen un valor de salida. Se podría tener un procedimiento que borre todos los clientes morosos de la base de datos de nuestra aplicación: El proceso finalizará y ningún resultado final será devuelto por el procedimiento ejecutado.

Parámetros de un módulo

Los parámetros son los datos de entrada para un módulo (función o procedimiento), datos de los cuales se alimentan para poder ejecutar su proceso asociado. El valor de los parámetros puede ser de cualquier tipo de dato: En algunos lenguajes de programación se requerirá declarar dichos tipos para cada parámetro, en otros lenguajes NO se declaran. Todo parámetro tiene un nombre identificador: será un nombre pertinente elegido por el programador.

Por otra parte, un módulo puede tener:

- Ningún parámetro.
- 1 parámetro.
- 2 ó más parámetros.

Los módulos sin parámetros no requieren que se les provea ninguna información para poder ejecutarse. Un módulo sin parámetros tiende a no ser dinámico, dado que siempre ejecutará la misma acción. Ejemplo: Un módulo que obtenga la fecha actual: NO requerirá de parámetros, ya que no debe proveerse ningún dato para poder obtener dicha fecha. A la acción de agregar parámetros a un módulo se la llama parametrización, con lo cual el módulo gana valores de entrada.

Por otro lado, los módulos con excesivos parámetros a menudo es una mala señal, dado que si necesitan una gran cantidad de datos para ejecutar su tarea asociada, podría ocurrir que dicho módulo “haga demasiado”: Su responsabilidad es muy grande. En estos casos siempre es viable crear sub-módulos, más específicos, de manera de delegar tareas específicas a otras nuevas piezas de la aplicación.

Observación: Recuerde también que un parámetro puede ser de tipo vector, por lo cual pueden proveerse múltiples datos al módulo pasando un argumento de tipo vector, al invocar al módulo en cuestión.

Concepto de argumento

El argumento es el valor específico que se le provee a un parámetro, para una determinada instancia de ejecución. Un módulo que obtenga el saldo a la fecha de uno de nuestros clientes requerirá que se le provea el nombre/ID del cliente: ‘YPF’ , ‘Cencosud’ ,‘Arcor’, etc; los cuales serán los argumentos para el parámetro del módulo en cuestión.

Declaración de funciones

En muchos lenguajes y en Javascript , la declaración será muy similar a la siguiente:

```
function nombrefuncion(parámetros){
```

```
    // Proceso a ejecutar
    ...
    return  valor final
}
```

Para que una función retorne un valor, se usará la palabra reservada “return”, el cual devuelve un resultado final y da por finalizado el proceso de la función. Un ejemplo:

```
function sumar(v1,v2){
    return v1 + v2;
}
```

Se trata de una función con 2 parámetros, el cual retorna la suma de sus argumentos, los cuales claramente deberán ser de tipo numérico.

Llamado / invocación de funciones:

Para ejecutar una función ya declarada; se deberá escribir el nombre de la función (con sus respectivos argumentos), en un determinado punto del programa (en el programa principal o bien dentro de otro módulo).

```
sumar(5,8);
```

Lo cual ejecutará la función, obteniendo un valor de retorno: En este ejemplo: “13”. El valor de retorno puede ser impreso en pantalla:

```
console.log('La suma es de: ' + sumar(5,8));
```

O incluso puede guardarse en una variable:

```
var suma = sumar(5,8);
```

En caso de ser necesario recuperar este valor posteriormente.

Declaración de procedimientos:

En lenguajes que incorporen procedimientos (Ejemplos: Pascal, MySQL); éstos suelen declararse con la palabra reservada “procedure”, seguida de su nombre identificador.

```
procedure identificador(parámetros){
            // Proceso a ejecutar.
}
```

Ejemplo:

```
procedure borrarCliente(idCliente){
          // Proceso de borrado aquí
}
```

Los procedimientos, al igual que las funciones, admiten en forma opcional el uso de parámetros (ya sea 1 ó múltiples), los cuales proveerán datos necesarios para la ejecución de dicho procedimiento. Un procedimiento siempre podrá a su vez invocar (hacer uso) de otro procedimiento de la aplicación, o bien hacer uso de una función.

Llamado / invocación de procedimientos:
No se presentan diferencias respecto al llamado de una función: Se procederá a escribir el nombre identificador del procedimiento + sus respectivos argumentos.

Módulos pre-construidos:

Los módulos pre-construidos son todos aquellos que no requieren ser implementados por el programador: Vienen ya incorporados por el propio lenguaje de programación en uso. Este tipo de módulo facilita la labor del programador, dado que él no tendrá que crear todas las funcionalidades partiendo de cero: El propio lenguaje colabora con tareas/problemas habituales. Para hacer uso de un módulo pre-construido bastará con invocarlo, escribiendo su nombre identificador con sus respectivos argumentos.

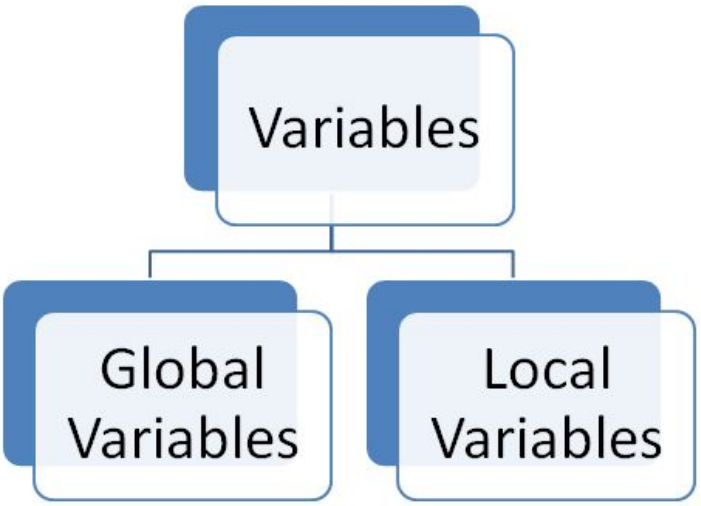
A menudo encontramos módulos pre-construidos que permitan:

- Obtener la cantidad de elementos de un vector.
- Ordenar un vector.
- Averiguar si un número entero es par o impar.
- Verificar la longitud de un string.
- Etc.

En caso de no haber un módulo pre-construido que resuelva un problema concreto que necesitamos implementar en nuestra aplicación, entonces en ese momento se deberá proceder a programar un módulo propio, utilizando las herramientas nativas que provee el lenguaje. Ejemplos de modulos pre-construidos en Javascript serían las funciones :

Number()	Transforma un valor a formato numérico
isNaN()	Verifica si un valor no es numérico
indexOf()	Verifica si cierto substring se encuentra dentro de cierto string

Ámbito de las variables



Las variables pueden clasificarse como locales o bien como globales, dependiendo del contexto en donde hayan sido declaradas.

- **Variables locales:**

Son aquellas variables declaradas dentro de un módulo (función y/o procedimiento):
Su alcance (entorno) será el propio módulo: por fuera de él, la variable NO será reconocida.

```
function calcular()
{
            var local = 'soy local';

            ....

            return ...
}

calcular();
console.log(local); //Error : undefined variable local
```

En el ejemplo anterior, se ejecuta la función, y luego se intenta obtener el valor de la variable "local", por fuera de la función "calcular": En ese caso se obtendrá un error, emitiendo un mensaje, en muchos lenguajes similar a: "undefined variable".

En los casos que un módulo "A" requiera transferir el valor de una variable local a otro módulo "B", siendo "B" invocado por "A", siempre podrá pasarse un valor a través de los parámetros que disponga el módulo "B", pudiendo ser entonces dicho valor la variable local de "A".

Ejemplo:

```
function modulo(){
            var local = 4;
            modulo_auxiliar(local);
            ...
}
```

- **Variables globales:**

Son aquellas variables declaradas fuera de un módulo de nuestra aplicación.
Estas variables tendrán un alcance total a todos los puntos de la aplicación, inclusive los propios módulos.

Ejemplo:

```
var global = 'soy global';
function procesar()
{
            ....

            global = 'Cambio de valor';

            ....
}
```

Los módulos pueden hacer un uso directo de las variables globales; aunque esta práctica de programación podría conllevar al difícil seguimiento de la aplicación ya que la variable global pudo haber sido definida en cualquier punto del código fuente, el cual puede no conocerse con precisión.

Siempre será una buena práctica de programación usar la menor cantidad de variables globales posibles, trabajando con variables locales, empleadas en módulos.

Por supuesto, siempre habrán casos en particular en los cuales las variables globales serán de utilidad para la aplicación que nos encontremos programando.

Concluyendo, por regla general, podemos decir que:

- Siempre será preferible trabajar con módulos específicos con sus variables propias (locales), que trabajar con aplicaciones que no modularizan, las cuales utilizarán exclusivamente variables de entorno global.

- Las variables globales atentan contra la modularidad de las aplicaciones.

ANEXO JavaScript:

JavaScript dispone del módulo Date, el cual posee distintos métodos para obtener información como:

- **getDate()** Número del día del mes (1 a 31)
- **getMonth()** Número del mes (0 a 11)
- **getFullYear()** Año (4 dígitos)
- **getDay()** Número del día en la semana (0⟶ Domingo, 1 ⟶ Lunes ...)
- **getHours()** Hora del día (0 a 23)
- **getMinutes()** Minutos de la hora (0 a 59)
- **getSeconds()** Segundos de la hora (0 a 59)
- **etc.**

Podemos entonces, por ejemplo, obtener el año actual de la siguiente manera:

```
var fecha          = new Date();
var dia            = fecha.getDate();
var mes            = fecha.getMonth() + 1; // comienza desde
cero..
var anio           = fecha.getFullYear ();
var hora           = fecha.getHours();
var minutos        = fecha.getMinutes();
var segundos       = fecha.getSeconds();
```

De ser necesario, podemos guardar el año en una variable.

También dispone de la función **setInterval** se utiliza para ejecutar otra función X cada intervalo regulares de N milisegundos.

```
setInterval(Function nombrefuncion,milisegundos);
//(Siendo milisegundos, un número entero, mayor a CERO.
```

Ejemplo :

```
var edad = 28;
function aumentarEdad(){
      edad++;
      console.log(edad);
}
setInterval(aumentarEdad,2000);
```

```
//28,29,30...cada 2 segundos
```

Nótese que en este caso no “invocamos” a la función usando sus paréntesis de ejecución, sino que solamente la nombramos para que la “ejecute” la función setInterval.

Habitualmente, la ejecución de la función X comienza ni bien la página ha terminado de cargar, lo cual conlleva a escribir la función setInterval en el evento **onload.** Ejemplo:

<body onload=“setInterval(nombrefuncion,3000);”>

Lo cual provocaría la ejecución de la función “nombrefuncion” cada 3000 milisegundos (3 segundos) ni bien la página ha cargado, infinitas veces. Un ejemplo práctico con setInterval podría ser crear una función “reloj” que obtiene la hora, minutos y segundos actuales (hh:mm:ss), la cual podemos ejecutarla en conjunto con setInverval de la siguiente manera:

<body onload=“setInterval(reloj,1000);”>

La función “reloj” se ejecuta cada 1 segundo. Consecuencia: Obtenemos un reloj digital para nuestra página.