



`/* Ciclos y métodos */`

Sesión conceptual 2





Inicio

{desafío}
latam_



15 minutos

- Utilizar ciclos anidados para resolver problemas.
- Crear métodos.
- Parametrizar métodos.
- Crear métodos con parámetros opcionales.
- Crear métodos con retorno.
- Conocer los retornos implícitos.
- Conocer los tipos de variable.
- Conocer el concepto de alcance.
- Diferenciar variables locales de variables globales.

Objetivo



Desarrollo

{desafío}
latam_



150 minutos

/* Introducción a ciclos anidados */

Escribiendo las tablas de multiplicar

Tabla de un número

- Supongamos que queremos mostrar una tabla de multiplicar. Por ejemplo la tabla del número 5:

```
10.times do |i|  
  puts "5 * #{i} = #{5 *  
i}"  
end
```

Tabla de todos los números

- ¿Cómo podríamos hacer para mostrar todas las tablas de multiplicar del 1 al 10?

```
10.times do |i|  
  10.times do |j|  
    puts "#{i} * #{j} =  
#{i * j}"  
  end  
end
```

/* Dibujando con ciclos anidados */

Ejercicio medio triángulo

- Crear el programa medio_triangulo.rb que reciba el tamaño del triángulo y dibuje el siguiente patrón:

```
ruby medio_triangulo.rb 5
```

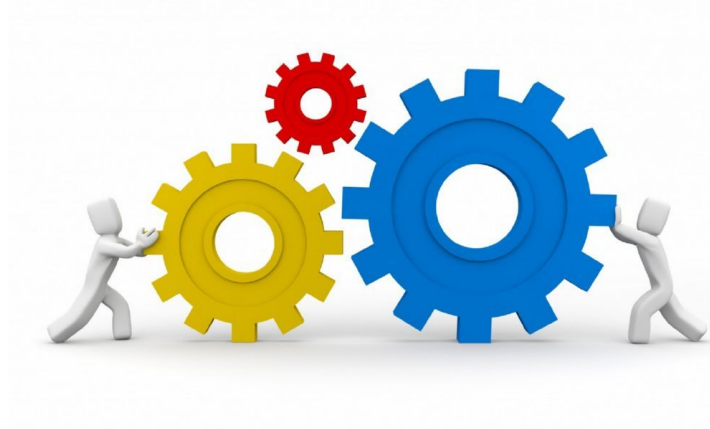
Ejercicio del cuadrado vacío

- Crear el programa cuadrado_hueco.rb que al ejecutarse reciba un tamaño y dibuje un cuadrado dejando vacío el interior.

/* Reutilizando código */

Métodos

- Los métodos nos permiten agrupar instrucciones y reutilizarlas.
- Podemos crear nuestros propios métodos, a esto le llamaremos definir.
- A utilizar métodos ya creados por nosotros, o por otras personas, le denominaremos llamar.



/* Creando métodos */

Definiendo un método

- En Ruby, un método se define utilizando la siguiente estructura:

```
def nombre_del_metodo # Definimos con def y un nombre.  
  # Serie de instrucciones que ejecutará el método.  
  # ...  
  # ...  
end # Terminamos de definir con end.
```

LLamando al método

- A los amigos, los llamamos por su nombre; a los métodos, de la misma forma. Si queremos llamar al método `imprimir_menu` simplemente escribiremos el nombre del método.

```
imprimir_menu  
imprimir_menu()
```

El orden de ejecución

En el momento que ejecutamos un código Ruby, este se lee de arriba a abajo. Ruby necesita leer las definiciones para luego poder utilizarlas.

- Llamar a un método antes de definirlo genera un error.

```
saludar()  
def saludar  
  puts "hola"  
end
```


/* Parametrizando métodos */

Creando un método con parámetro

- Crear un método que recibe un parámetro es sencillo. En la definición del método utilizaremos paréntesis para especificar los parámetros que debe recibir.

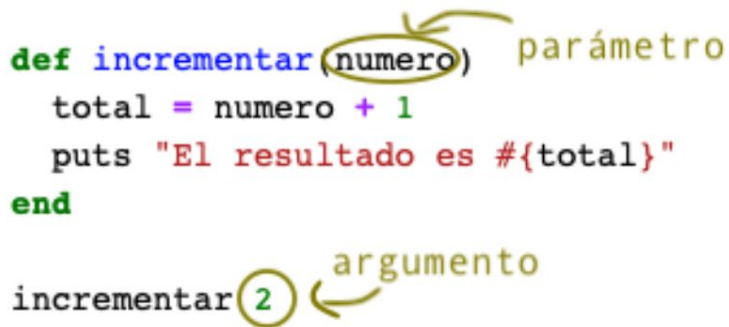
```
def incrementar(numero)
  total = numero + 1
  puts "El resultado es #{total}"
end
```

¿Qué es un argumento?

En el error leímos Wrong number of arguments pero ¿Qué es un argumento?

```
def incrementar(numero) parámetro
  total = numero + 1
  puts "El resultado es #{total}"
end

incrementar(2) argumento
```

The diagram illustrates the difference between a parameter and an argument in Ruby. In the function definition 'def incrementar(numero)', the word 'numero' is circled in green and labeled 'parámetro' with a green arrow. In the function call 'incrementar(2)', the number '2' is circled in green and labeled 'argumento' with a green arrow.


- Las variables, en la definición de un método, se denominan **parámetro**.
- Los objetos que pasamos, al llamar al método, se denominan **argumento**.

Métodos con parámetros opcionales

Para crear un método que recibe parámetros de forma opcional tenemos que especificar qué debe hacer Ruby cuando el parámetro no se especifica.

```
def incrementar(numero, cantidad = 1)
  total = numero + cantidad
  puts "El resultado es #{total}"
end
```

Un valor opcional
tiene un valor asignado
por defecto



```
incrementar 2, 1 # 3
incrementar 2 # 3
incrementar 2, 2 # 4
```

Desafío

Crear el programa `validar_edad.rb` que contenga el siguiente código pero que cumpla las siguientes condiciones:

- Modificar el método para que reciba la edad.
- Llamar al método 3 veces, con edades generadas al azar.

/* Retorno */

El retorno

- Los métodos pueden recibir parámetros y pueden devolver un valor. A este valor se le conoce como retorno. Todos los métodos tienen un retorno, en algunos casos es implícito, y en otros, explícito.

Retorno explícito:

```
def fahrenheit(f)
  celsius = (f + 40) / 1.8 - 40
  return celsius
end
```

Retorno implícito:

```
def fahrenheit(f)
  celsius = (f + 40) / 1.8 - 40
end
```

Return sale del método

Todo lo que viene después de la instrucción return es ignorado.

```
def prueba
  x = 'mensaje que no se muestra'
  return # Punto de salida
  puts x
end

prueba # => nil
```


/* Alcance de variables */

Tipos de variable

En Ruby existen 4 tipos de variable:



Globales

Locales

De instancia

De clase

El alcance

El alcance, o Scope en inglés, es desde dónde podemos acceder a una variable.

- Alcance de una variable local.
- Los parámetros también cuentan como variables locales.
- No importa el orden de declaración.

El espacio principal de trabajo recibe el nombre de main.

```
# Esto está siendo definido en el ambiente 'main'
```

```
nombre = 'Homero Simpson'
```

```
edad = 40
```

```
def cualquier_metodo
```

```
# Esto está siendo definido en un ambiente nuevo: el del método
```

```
# Aquí no existen las variables nombre ni edad
```

```
palabra = 'diez'
```

```
numero = 10
```

```
end
```

variables locales

- `address` es una variable local, cuyo alcance es `'main'`.
- `occupation` es una variable local, cuyo alcance es el método `presentar`.
- `address` no existe dentro del método `presentar`.
- `occupation` no existe fuera del método `presentar`.

```
defined? edad
```

```
# => "local-variable"
```

Variables globales

Las variables globales, como su nombre lo indica, pueden ser accedidas desde todos los espacios.

```
$continente = 'Sudamérica'
```

```
# => "Sudamérica"
```

```
defined? $continente
```

```
#=> "global-variable"
```

El problema de las variables globales

Las variables globales son consideradas una mala práctica ya que hacen muy fácil romper un programa por error.

¿Cómo se rompe un programa con variables globales?

- En este aspecto es sencillo que alguien llame por error a una variable de la misma forma que otra persona la llamó y cualquier cambio puede romper todo el código.



Quiz

{desafío}
latam_



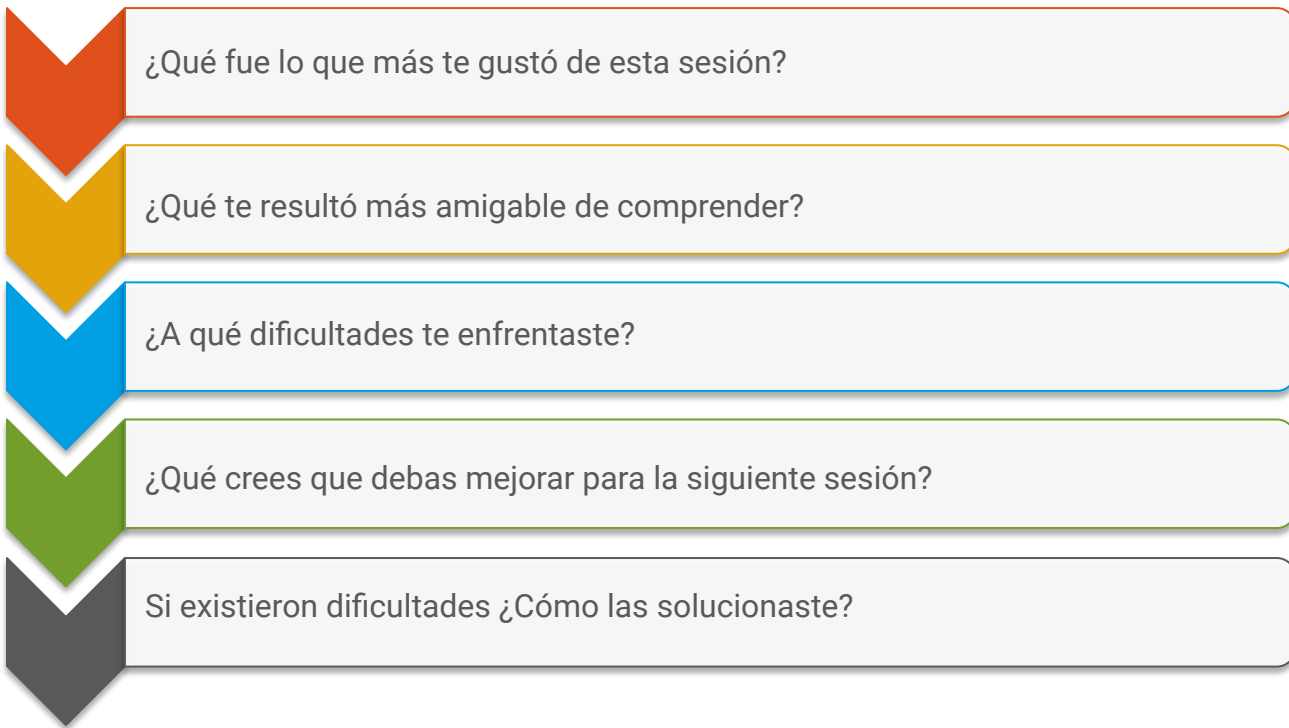


Cierre

{desafío}
latam_



15 minutos



¿Qué fue lo que más te gustó de esta sesión?

¿Qué te resultó más amigable de comprender?

¿A qué dificultades te enfrentaste?

¿Qué crees que debas mejorar para la siguiente sesión?

Si existieron dificultades ¿Cómo las solucionaste?



*Academia de
talentos digitales*

www.desafiolatam.com



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam