

Ciclos y métodos

Lo que debes saber antes de comenzar esta unidad

Un algoritmo es una serie de pasos para resolver un problema.



Estos pasos pueden ser:

- Entradas de datos
 - get
 - ARGV[0]
- Salidas de datos
 - puts
 - print
- Asignaciones
 - a = 2

- Comparaciones
 - `a == 3`
- Operaciones matemáticas y precedencia.
 - `a + 1 * 2 - 1`
- Operaciones lógicas
 - `a && !b`
- Decisiones
 - `if a == 2`
- Otros
 - creación de un número al azar

Tipos de datos

- String: "hola"
- Integer: 1
- Boolean: true
- Nil: nil

Transformando datos

- `to_i` para pasarlo a integer
- `to_f` para pasarlo a float.
- `to_s` para pasarlo a string.

Utilizando estas instrucciones aprendimos a resolver distintos tipos de problemas:

- Resolver una ecuación matemática
 - Permitir al usuario ingresar valores
- Crear una calculadora

Capítulo: Ciclos

Objetivos

- Conocer los ciclos y sus posibles aplicaciones en la programación.
- Leer y transcribir diagramas de flujo con iteraciones a código Ruby.
- Validar una entrada de datos.
- Crear un menú de opciones.

Introducción a ciclos

Los ciclos son instrucciones que nos permiten repetir la ejecución de una o más instrucciones.

```
Mientras se cumple una condición:  
  Instrucción 1  
  Instrucción 2  
  Instrucción 3
```

Repetir instrucciones es la clave para crear programas avanzados.

Usos de ciclos

Los posibles usos de ciclos en algoritmos son infinitos, nos permiten recorrer colecciones de datos o espacios de búsqueda.

Parte importante de aprender a programar es entender cómo utilizarlos correctamente y desarrollar las habilidades lógicas para entender cuándo utilizarlos.

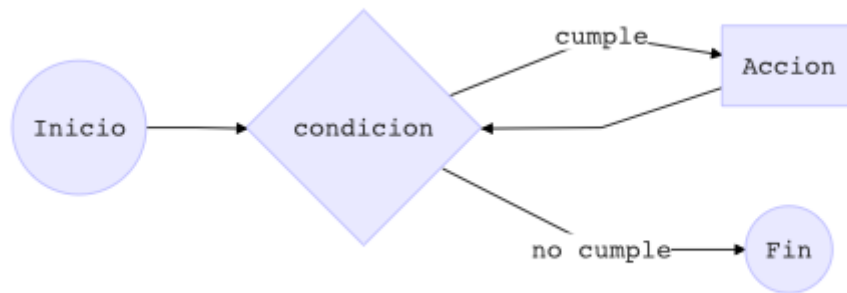
En esta unidad estudiaremos distintos problemas que se resuelven con ciclos y que, si bien no siempre son utilizados en la industria, nos ayudarán a desarrollar las habilidades lógicas que necesitamos para ser buenos programadores.

WHILE

La instrucción `while` nos permite ejecutar una o más operaciones **mientras** se cumpla una condición. Su sintaxis es la siguiente:

```
while(condition)  
  # Código que se ejecuta  
end
```

While paso a paso



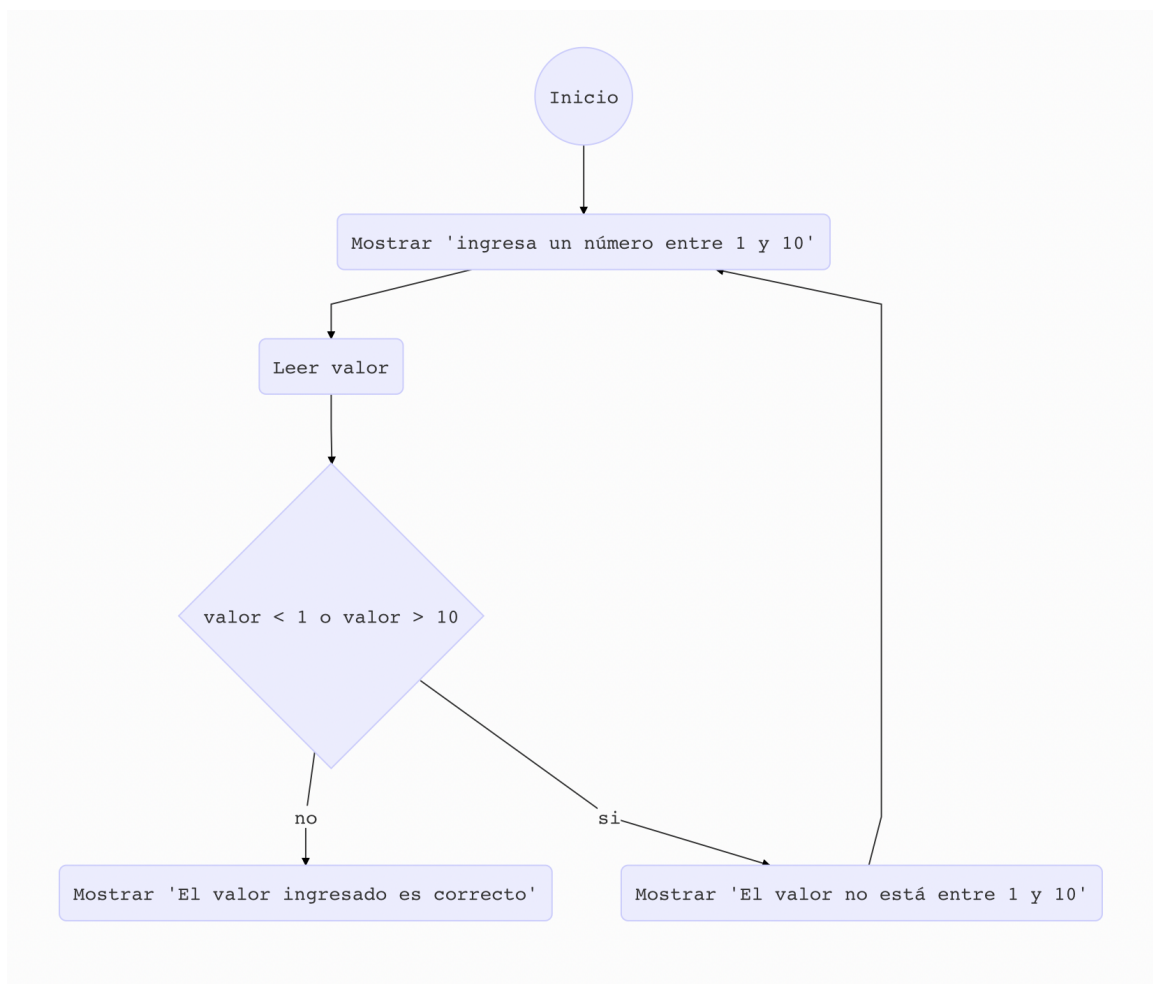
1. Se evalúa la condición; si es `true` , ingresa al ciclo.
2. Se ejecutan, secuencialmente, las instrucciones definidas dentro del ciclo.
3. Una vez ejecutadas todas las instrucciones se vuelve a evaluar la condición:
 - Si se evalúa como `true` : vuelve a repetir
 - Si se evalúa como `false` : sale del ciclo

Salida del ciclo

Un algoritmo es una secuencia **FINITA** de pasos para resolver un problema. En algún momento alguna de las instrucciones ejecutadas al interior del ciclo debe lograr que la condición deje de cumplirse.

Validación de entrada de datos utilizando while

Un ejemplo común para comenzar a estudiar los ciclos es validar la entrada de un dato, es decir, que este dato **cumpla un criterio**. Podemos, por ejemplo, validar que el usuario ingrese un número entre 1 y 10:



Para escribir este código crearemos el programa `primer_ciclo.rb`

```
puts 'Ingresa un número entre 1 y 10: '
num = gets.to_i

while num < 1 || num > 10
  puts 'El número ingresado no está entre 1 y 10'
  puts 'Ingresa un número entre 1 y 10: '
  num = gets.to_i
end

puts "El número ingresado fue #{num}."
```

¿Por qué es necesario declarar dos veces el ingreso de datos por parte del usuario?

Si no solicitamos al usuario el ingreso del dato antes del ciclo entonces, cuando el flujo llegue a la evaluación en el ciclo `while`, la variable `num` no estará definida, no ingresaremos nunca al ciclo y obtendremos un error.

```
puts 'Ingresa un número entre 1 y 10: '
while num < 1 || num > 10
  puts 'El número ingresado no está entre 1 y 10'
  puts 'Ingresa un número entre 1 y 10: '
  num = gets.to_i
end

puts "El número ingresado fue #{num}."
# undefined local variable or method `num' for main:Object (NameError)
```

¿Qué es num?

Until

Existe una instrucción que es la inversa de `while`

En decir, `while i < 0` es lo mismo que `until i >= 0`

`while` es mucho más utilizado que `until`, pero es bueno conocerlo por si lo encontramos en algún ejemplo.

Aviso sobre la instrucción gets

Durante el desarrollo de programas en general **no** utilizaremos `gets`. Al crear programas utilizaremos argumentos con `ARGV` o valores al azar con `rand`, o traeremos los datos desde archivos.

El uso de esta instrucción `gets` bloquea el programa hasta que el usuario ingresa un valor, esto complica el uso y evaluación de scripts. De todas formas podemos agradecer que `gets` nos ayudó a realizar ejercicios simples con ciclos.

Ejercicio: Validación de password

Objetivos

- Crear un diagrama de flujo con ciclos
- Crear un programa llamado `password_validation.rb` que valide la contraseña de un usuario

Podemos utilizar la misma idea de validación aprendida para impedir al usuario entrar hasta que ingrese la contraseña "password". Recuerda hacer el diagrama de flujo antes del código. Puedes utilizar el del capítulo anterior como base.

Solución

```
puts 'Ingresa su contraseña:'  
password = gets.chomp  
  
while password != 'password'  
  puts 'La contraseña es incorrecta'  
  puts 'Ingresa su contraseña:'  
  password = gets.chomp  
end  
  
puts "La contraseña ingresada es correcta!"
```

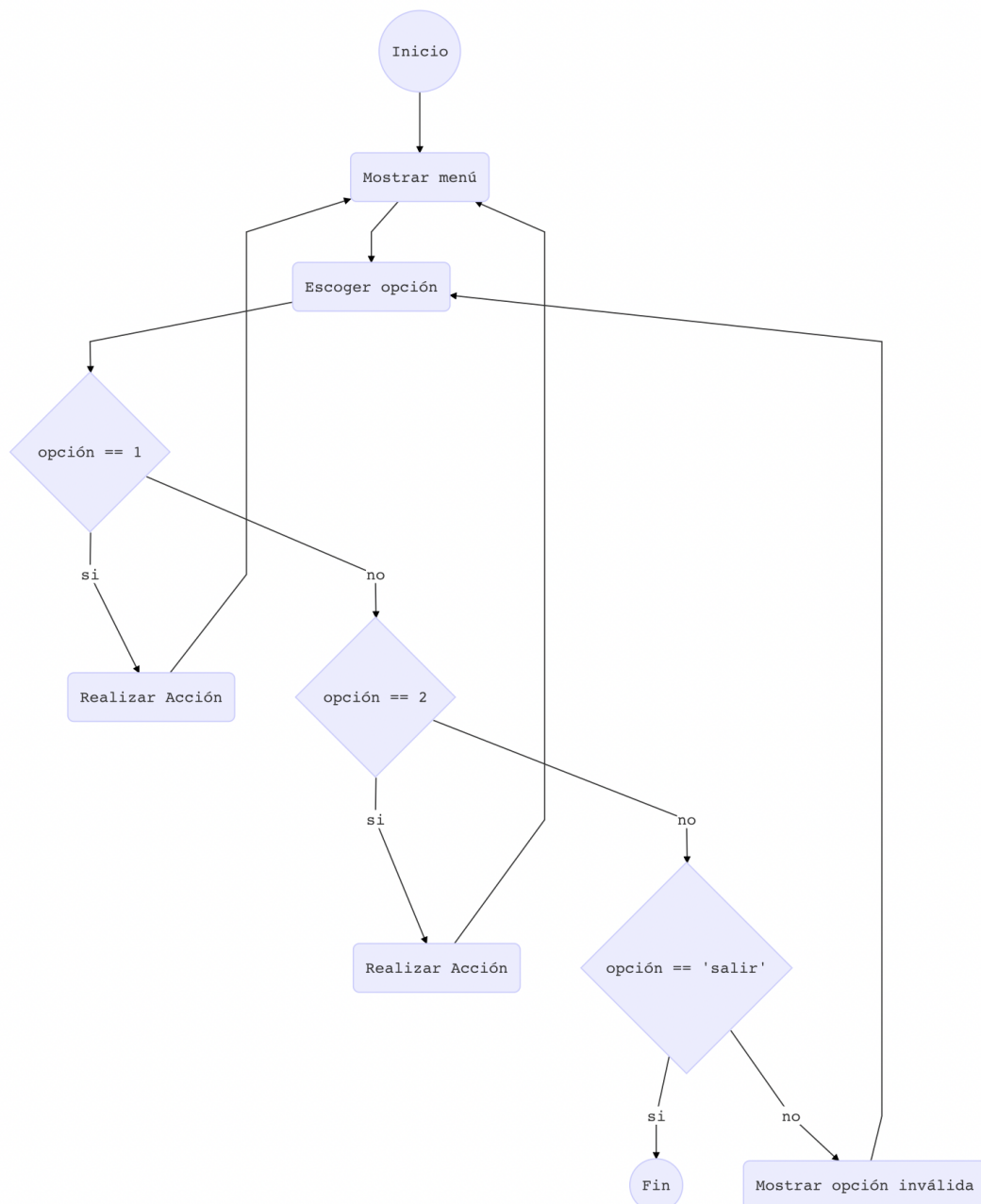
Ejercicio menú de opciones

Podemos implementar de forma sencilla un menú de opciones para el usuario. La lógica es similar a la de la validación de entrada.

- Se muestra un texto con opciones.
- El usuario tiene que ingresar una opción válida -> validación de entrada.
- Si el usuario ingresa la opción 1 mostramos un texto.
- Si el usuario ingresa la opción 2 mostramos otro texto.
- Si el usuario ingresa la opción "salir" terminamos el programa.

Desarrolla el diagrama de flujo de la solución y luego implementa el algoritmo. En el próximo capítulo revisaremos la solución

Diagrama de flujo del Menú



El código para un menú

```
opcion_menu = 'cualquier valor'
while opcion_menu != 'salir' && opcion_menu != 'Salir'
    # Mostrar menú
    puts 'Escoge una opción:'
    puts '-----'
    puts '1 - Acción 1'
    puts '2 - Acción 2'
    puts 'Escribe "salir" para terminar el programa'

    puts 'Ingresa una opción:'
    opcion_menu = gets.chomp

    if opcion_menu == '1'
        puts 'Realizando acción 1...'
    elsif opcion_menu == '2'
        puts 'Realizando acción 2...'
    elsif opcion_menu == 'salir' || opcion_menu == 'Salir'
```



```
    puts 'Saliendo...'
  else
    puts 'Opción inválida'
  end
end
```

Capítulo: Ciclos y contadores

Objetivos

- Conocer el concepto de iteración.
- Contar la cantidad de veces que un programa está dentro de un ciclo.
- Realizar programas donde el usuario ingrese múltiples datos hasta que decida detenerse.

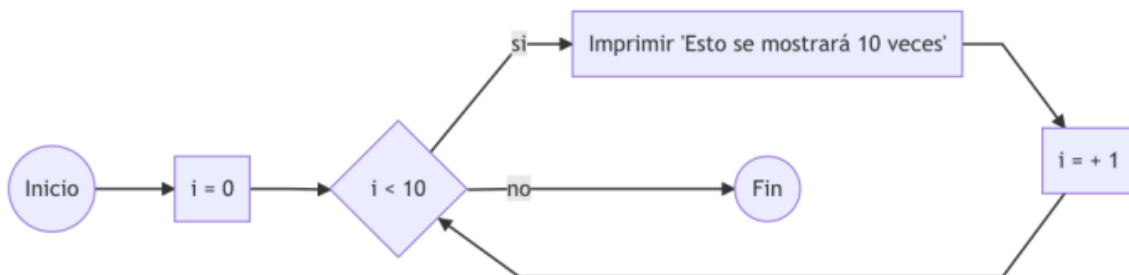
Introducción

En los capítulos anterior resolvimos ejercicios de ciclos donde el fin del ciclo estaba determinado por el ingreso de un dato por parte del usuario.

En este capítulo resolveremos problemas que requieren una cantidad determinada de ciclos. Por ejemplo un programa que cuente de cero a diez o un problema de sumatorias.

Iterar

Iterar es dar una vuelta al ciclo. Hay muchos problemas que se pueden resolver iterando. Por ejemplo repetir un mensaje 10 veces o contar desde cero hasta 10.



Contando con while

```
i = 0
while i < 10
  puts "Esto se mostrará 10 veces" # Código que queremos repetir.
  i += 1 # IMPORTANTE
end
```

La instrucción `puts "Esto se mostrará 10 veces"` se repetirá hasta que la variable `i` alcance el valor `10`. Para entonces, la comparación de la instrucción `while` se evaluará como `false` y saldremos del ciclo.

En programación, es una convención ocupar una variable llamada `i` como variable de iteración para operar en un ciclo.

IMPORTANTE: Si no aumentamos el valor de la variable `i` entonces nunca llegará a ser igual o mayor a 10, por ende, la comparación nunca se evaluará como `false` y entraremos en un ciclo infinito.

¿Qué significa `i+= 1`?

`+=` es un operador de asignación, muy similar a decir `a = 2` pero la diferencia es que con `+=` estamos diciendo el valor anterior más 1.

```
a = 2
a += 2 #aquí el valor de 'a' aumentó en dos y fue almacenado nuevamente en 'a'

puts a
# 4
```

Esto es lo mismo que escribir:

```
a = 2
a += 2
puts a
```

Operadores de asignación

La siguiente tabla muestra el comportamiento de los operadores de asignación:

Operador	Nombre	Ejemplo	Resultado
<code>=</code>	Asignación	<code>a = 2</code>	a toma el valor 2
<code>+=</code>	Incremento y asignación	<code>a += 2</code>	a es incrementado en dos y asignado el valor resultante
<code>-=</code>	Decremento y asignación	<code>a -= 2</code>	a es reducido en dos y asignado el valor resultante
<code>*=</code>	Multiplicación y asignación	<code>a *= 3</code>	a es multiplicado por tres y asignado el valor resultante
<code>/=</code>	División y asignación	<code>a /= 3</code>	a es dividido por tres y asignado el valor resultante

Ejercicio: La bomba de tiempo

Crearemos un algoritmo sencillo que realice una cuenta regresiva de 5 segundos.

Contar de forma regresiva es muy similar, solo debemos comenzar desde el valor correspondiente e ir disminuyendo su valor de uno en uno.

```
i = 5
while(i > 0)
  i -= 1
  puts i
end
```

```
4
3
2
1
0
```

Contando segundos

Existe una instrucción llamada `sleep` que nos permite esperar un tiempo determinado antes de continuar.

```
i = 5
while(i > 0) # Cuando llegue a cero terminamos.
  i -= 1 # En cada iteración descontamos 1.
  puts i
  sleep 1
end
```

```
4
3
2
1
0
```

Cuidado con las condiciones de borde

No es lo mismo

```
i = 5
while(i > 0)
  i -= 1
  puts i
  sleep 1
end
```

Que:

```
i = 5
while(i >= 0) # el valor cero cumple la condición
  i -= 1
  puts i
  sleep 1
end
```

Capítulo: Ciclos y sumatorias

Objetivos

- Conocer las operaciones de sumatoria.
- Crear diagramas de flujo de problemas de sumatoria.
- Escribir en Ruby el código de una sumatoria.
- Conocer la diferencia de un contador y de un acumulador.

Motivación

En este capítulo aprenderemos a crear programas que, además de utilizar ciclos, operen sobre otra variable.

Esto nos permitirá resolver diversos tipos de problemas, la mayoría de ellos corresponde a problemas del tipo matemáticos pero son aplicables a múltiples contextos.

Introducción a sumatorias

Para algunos -que no gustan de las matemáticas- el término sumatoria puede sonar algo intimidante. Sin embargo, para resolver una sumatoria, solo necesitamos saber una cosa: sumar.

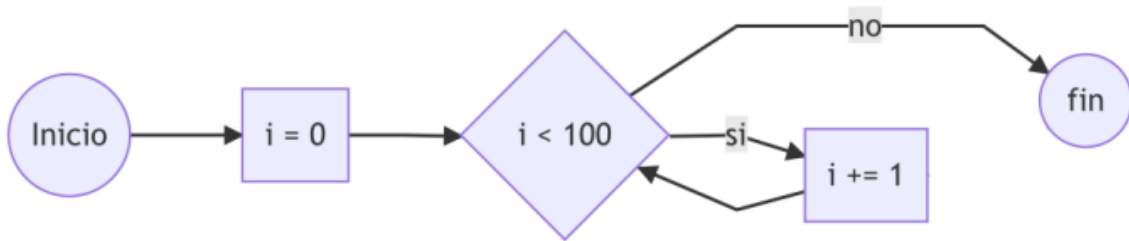
Sumando números

$$1 + 2 + 3 + \dots + 100 = ?$$

La sumatoria consiste en sumar todos los números de una secuencia. Por ejemplo: sumar todos los números entre 1 y 100. Esto no solo sirve para resolver ecuaciones matemáticas, sino también para que generemos las habilidades de abstracción necesarias para resolver diversos problemas.

Resolver esto es muy similar a contar las cien veces, pero además de contar, vamos ir guardando la suma **de cada iteración**.

Comencemos desde la base del código anterior



```

i = 0
while i < 100
  i += 1
end
  
```

Luego, si queremos guardar la suma en cada iteración, necesitamos una variable para ir guardando los datos.

```

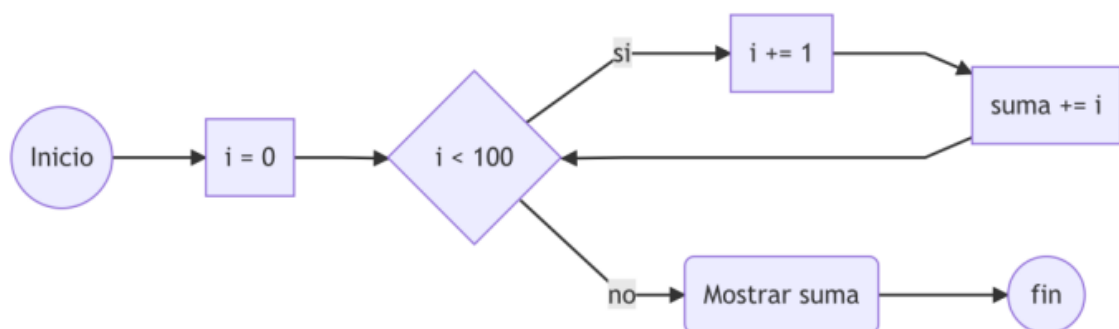
i = 0
while i < 10
  i += 1
  suma += i
end
  
```

Todavía nos falta un detalle para que funcione. No podemos sumarle algo a una variable que no existe; las variables con las que contamos o sumamos deben ser, primero, **inicializadas**.

```

i = 0
suma = 0
while i < 10
  i += 1
  suma += i
end
  
```

Veamos el diagrama de flujo para reforzar lo aprendido.



La instrucción `suma = suma + i` es la encargada de aumentar el valor de la variable `suma` en cada iteración.

Este comportamiento, sumar y almacenar secuencialmente valores variables, se conoce como la implementación de un **acumulador**. En este ejemplo, la variable `suma` está acumulando la suma de los valores en cada iteración.

Resumen del capítulo

En este capítulo aprendimos sobre contadores y acumuladores.

Tanto los contadores como acumuladores son **ampliamente utilizados** en programación.

- Contador: Aumenta de 1 en 1
 - `cont = cont + 1`
 - `cont += 1`
- Acumulador: Aumenta en función a `valor`
 - `acu = acu + valor`
 - `acu += valor`

Desafío `suma_n.rb`

- **`suma_n.rb`**: El usuario ingresa un número, se muestra la suma de todos los números de 1 hasta ese número.

uso:

`suma_n.rb 100`

```
5050
```

Desafío `pares.rb`

Crear el programa `pares.rb` donde se sumen únicamente los números pares dentro del ciclo entre 0 y un número ingresado por el usuario al momento de cargar el programa.

uso:

`pares.rb 100`

```
2550
```

Solución

```
suma_n.rb
```



```

# Valores iniciales
limit = ARGV[0].to_i
i = 0
suma = 0

# Iteracion
while i < limit
  i += 1
  suma += i
end

# Resultado
puts suma

```

pares.rb

```

# Valores iniciales
limit = ARGV[0].to_i
i = 0
suma = 0

# Iteracion
while i < limit
  i += 2
  suma += i
end

# Resultado
puts suma

```

Desafío: Generador de listas en HTML

Se pide crear el programa `generador_li.rb` donde el usuario ingrese un número como argumento y se genere una lista de HTML con esa cantidad de ítems.

Uso: `ruby lista_html.rb 5`

```

<ul>
  <li> 1 </li>
  <li> 2 </li>
  <li> 3 </li>
</ul>

```

Pistas:

- Puedes tabular con `"\t"`
- Puedes hacer un salto de línea con `"\n"`
- Hay elementos que están antes del ciclo y otros después.

Solución al generador de listas

Lo primero es identificar las partes que se repiten y las que no. En este caso, las etiquetas `` son los elementos que se repiten; en cambio `` y ``, solo aparecen al principio y al final.

Preocupémonos primero de la parte que se repite. Para esto crearemos el programa `lista_html.rb`, y dentro escribiremos:

```
html = ""
items = ARGV[0].to_i
i = 0
while i < items
  i += 1
  html += "<li> item #{i} </li>\n"
end
puts html
```

Si llamamos a nuestro programa con: `ruby lista_html.rb 3` obtendremos:

```
<li> item 1 </li>
<li> item 2 </li>
<li> item 3 </li>
```

El siguiente paso es agregar el principio y el final

```
html = "<ul>\n"
items = ARGV[0].to_i
i = 0
while i < items
  i += 1
  html += "<li> item #{i} </li>\n"
end
html += "</ul>"
puts html
```

Finalmente, solo nos falta agregar las tabulaciones

```
html = "<ul>\n"
items = ARGV[0].to_i
i = 0
while i < items
  i += 1
  html += "\t<li> item #{i} </li>\n"
end
html += "</ul>"
puts html
```

Capítulo: Otras instrucciones para ciclos

Objetivos:

- Aprender la sintaxis de otros elementos que permiten iterar dentro de Ruby.
- Conocer la ventaja y desventaja de estas instrucciones.
- Conocer los rangos.
- Conocer los bloques.

Motivación

Además de `while` existen otras dos instrucciones muy útiles para iterar.

- `for`
- `times`

Algunas situaciones que estudiaremos serán más sencillas de resolver utilizando estas instrucciones.

Ciclos con la instrucción For

La instrucción `for` nos permite iterar en un rango. Podemos, por ejemplo, iterar de 1 a 10 con la siguiente sintaxis:

```
for i in 1..10
  puts "Iteración #{i}"
end
```

Recordemos que es convención utilizar `i` como variable de iteración.

Rangos

`1..10` es un tipo de dato bien especial, llamado rango.

```
(1..10).class
# Range
```

Para crear un rango podemos utilizar caracteres o enteros.

```
for i in 'a'..'z'  
  puts i  
end
```

Disminuir es ligeramente más complejo.

```
for i in 10.downto(1)  
  puts "hola #{i}"  
end
```

Ventaja de `for`

La principal ventaja de `for` es que nos podemos olvidar de implementar un contador, ya que la variable de iteración aumenta en cada iteración de forma automática.

Desventaja de `for`

La desventaja es que es menos flexible, porque no podemos manipular el incremento de manera personalizada.

El ciclo Times

El método `times` que podemos traducir como a `veces` nos permite iterar de una forma muy sencilla.

```
5.times do  
  puts "repitiendo"  
end
```

Un comportamiento interesante: ¡podemos imprimir el número de iteración en el que estamos trabajando!

```
5.times do |i|  
  puts "repitiendo: #{i}"  
end
```

Recordemos que el nombre `i` para la variable es sólo una convención, podemos utilizar el nombre que más nos acomode.

Introducción a bloques

Tanto `times` como muchos otros métodos pueden recibir **bloques**.

Este concepto es **MUY** importante en Ruby.

Profundizaremos en él más adelante, sin embargo en este punto necesitamos entender su sintaxis.

Un bloque se puede escribir de dos formas:

1º Forma para escribir un bloque: Con `do` y `end` :

```
10.times do
  puts 'Repitiendo 10 veces'
end
```

2º Forma para escribir un bloque: Entre llaves `{ }`

```
10.times { puts 'repitiendo 10 veces' }
```

Esta segunda forma recibe el nombre de **inline** porque es ideal para escribir bloques de una sola línea.

Los bloques son closures

El concepto de **closure** viene del mundo matemático y su definición puede ser bien compleja de entender, por ejemplo si vemos la de wikipedia encontraremos:

Es una técnica para implementar ámbitos léxicos en un lenguaje de programación con funciones de primera clase.

En palabras sencillas

Todo lo que declaremos dentro de un bloque queda definido **solo** dentro del bloque

Ejemplo de declaración dentro de un bloque

```
10.times do |i|
  z = 0
end

puts z # undefined local variable or method `z' for main:Object
```

Esto solo aplica a las declaraciones y no a las asignaciones

```
z = 0
10.times do |i|
  z += 1
end

puts z
```

10

Si este tema no te quedó completamente claro no te preocupes, lo revisaremos más adelante en esta unidad cuando discutamos el concepto de alcance de variables. En este momento lo único importante es recordar que las variables declaradas dentro de bloques solo pueden ser accedidas dentro del mismo bloque.

Iterando con times y el índice

De la primera forma:

```
10.times do |i|
  puts i
end
```

De forma inline

```
10.times { |i| puts i }
```

Al igual que las variables definidas dentro del bloque, las variables de iteración tampoco existen fuera del ciclo

```
10.times do |i|
  puts "repitiendo: #{i}"
end

puts i

# repitiendo: 0
# ...
# ...
# repitiendo: 9
# undefined local variable or method `i' for main:Object (NameError)
```

Capítulo: Sumatorias y productorias

Objetivo:

- Leer expresiones de sumatorias
- Escribir código para resolver sumatorias

Introducción

Es muy frecuente para un programador tener que implementar fórmulas matemáticas.

Por ejemplo, al realizar algún cálculo específico del negocio, proyecciones de venta, o incluso algo más directo como trabajar en una simulación de algún área científica. Es por eso que tenemos que sentirnos cómodos como programadores viendo e implementando en código diversos tipos de fórmulas matemáticas.

$$\sum_{i=5}^{100} i = 5 + 6 + 7 + \dots + 100$$

El número inferior nos da el primer valor de la iteración, el de arriba el último, por eso en esta expresión el primer número de la suma es 5 y el último es 100.

En código sería:

```
suma = 0
for i in (5..100)
  suma += i
end
print suma
```

5040

No siempre sumaremos de uno en uno. Veamos la siguiente expresión.

$$\sum_{i=3}^9 2i = 2 * 3 + 2 * 4 + \dots + 2 * 9$$

Implementar esto en código también es bastante directo.

```
suma = 0
for i in (3..9)
  suma += 2*i
end
print suma
```

La expresión puede ser llegar a ser un poco más compleja, pero siempre es la misma idea.

$$\sum_{i=1}^{10} i^2 + 2i$$

```
suma = 0
for i in (1..10)
  suma += (i**2)+2*i
end
print suma
```

495

Productorias

Existe una expresión similar a las sumatorias, llamadas productorias. son exactamente lo mismo, pero los números se multiplican en lugar de sumarse

$$\prod_{i=1}^{10} i = 1 * 2 * 3 * \dots$$

```
producto = 1 # Es importante no inicializar el producto en 0, porque cualquier multiplicación por cero
              dará como resultado cero.
for i in (1..10)
  producto *= i
end
print producto
```

3628800

Capítulo: Introducción a patrones con ciclos

Objetivo:

- Reconocer patrones de repetición en un ciclo.
- Determinar si un número es par o impar
- Utilizar la paridad de un número para dibujar patrones

Introducción

Una de las dificultades más frecuentes -en principiantes- al momento de resolver problemas de ciclos es la de identificar/entender el patrón.

No entender un patrón de manera rápida e instantánea no nos hace menos inteligentes. A pesar de que hay personas que pueden resolver estos problemas de manera intuitiva, la mayoría de nosotros tuvo que aprender a resolverlos.

En este capítulo vamos a resolver problemas desde muy sencillos hasta complejos, identificando el patrón.

Desafío 1: Dibujando puntos

Crear el programa solo_puntos.rb que dibuje `n` puntos. Donde `n` es un valor ingresado por el usuario al momento de ejecutar el script.

Uso:

```
ruby solo_puntos.rb 5
```

```
resultado:  
*****
```

```
ruby solo_puntos.rb 1
```

```
resultado:  
*
```

Solución no recomendada

El primer intento para resolverlo podría ser utilizando instrucciones `if`:

```

n = ARGV[0].to_i
if n == 1
  puts '*'
elsif n == 2
  puts '**'
elsif n == 3
  puts '***'
elsif n == 4
  puts '****'
elsif n == 5
  puts '*****'
end

```

Sin embargo, la solución es bastante limitada. ¿Qué sucedería si el usuario ingresa el valor 6? ¿o 7? ¿o 100?. ¿Vamos a programar todas las opciones hasta 100?. Este tipo de problemas se resuelve mucho mejor con ciclos.

Solución 2

Para resolver el problema con ciclos debemos, simplemente, identificar el patrón. Si el usuario ingresa 1, se dibuja un asterisco; si el usuario ingresa 2, se dibujan 2 asteriscos. Si el usuario ingresa `n` entonces hay que dibujar `n` asteriscos .

```

n = ARGV[0].to_i
n.times do
  print '*' # Tenemos que utilizar `print` en lugar de `puts`
            # porque `puts` insertaría automáticamente un salto de línea
end

```

Solución 3

En Ruby es fácil lograr una solución sin ciclos, multiplicando `"*"` con `n`

```

n = ARGV[0].to_i
print '*' * n

```

Pero la solución con ciclos es la que tenemos que poder lograr.

Desafío 2: Puntos y números

Crear el programa `puntos_y_numeros.rb` que dibuje `N` números intercalados por puntos. Donde `N` es un valor ingresado por el usuario al momento de ejecutar el script.

Uso:

```
ruby solo_puntos.rb 5
```

```
resultado:  
0.2.4
```

ruby solo_puntos.rb 9

```
resultado:  
0.2.4.6.8
```

¿Cuál es el patrón?

La primera pregunta es cuál es el patrón, en este caso vemos que los números impares son los que son remplazados por puntos y los números pares se muestra el mismo número. Para este experimento consideraremos el cero como número par.

Si número es par => punto Si número es impar => número

¿Par o impar?

Hay dos formas de saber si un elementos es par.

- Utilizar el método `.even`
- Utilizar la operación resto (o módulo)

Utilizando el método `.even?`

En ruby existe un método muy sencillo llamado `.even?`

```
2.even? # => true  
3.even? # => false
```

Utilizando la operación resto

Pero existe otra forma un poco más flexible y muy utilizada en general que es calcular el resto de la operación. O sea, si al dividir un número por dos, la división es exacta, es par, y si queda un resto, entonces es impar.

```
5 % 2 == 0 # => false  
5 % 2 == 1 # => true
```

Solución

```
n = 5
n.times do |i|
  if i % 2 == 0 # Si es par
    print i
  else
    print '.'
  end
end
```

Desafío 3: Dibujando asteriscos y puntos

Crear el programa `asteriscos_y_puntos.rb` que dibuje **asteriscos y puntos intercalados** hasta `n`. Donde `n` es un valor ingresado por el usuario al momento de ejecutar el script.

Uso:

`ruby asteriscos_y_puntos.rb 3`

```
resultado:
*.*
```

`ruby asteriscos_y_puntos.rb 4`

```
resultado:
*.*.
```

`ruby asteriscos_y_puntos.rb 5`

```
resultado:
***.
```

Solución

Para resolver este ejercicio debemos iterar utilizando el índice, ya que debemos identificar si nos encontramos en una posición par o impar.

```
#n = ARGV[0].to_i
n = 8
n.times do |i|
  if i.even?
    print '*'
  else
    print "."
  end
end
```

Desafío 4

Crear el programa `dos_por_dos.rb` que dibuje el siguiente patrón de asteriscos y puntos intercalando hasta `n`. Donde `n` es un valor ingresado por el usuario al momento de ejecutar el script.

```
ruby dos_por_dos.rb 5
```

```
**..*
```

```
ruby dos_por_dos.rb 6
```

```
**..**
```

Siempre hay que partir estudiando la solución. En este caso el patrón se repite cada 4 caracteres
caracter 1 y 2 => * caracter 3 y 4 => .

Para resolver este tipo de patrones podemos ocupar nuevamente la operación resto (o módulo) y como el patrón se repite cada 4 caracteres entonces utilizaremos el resto de 4.

i	i%4
0	0
1	1
2	2
3	3
4	0
5	1
6	2
7	3
8	0
9	1
10	2
11	3
12	0

- Si $i\%4$ es 0 o 1 mostraremos un *
- En otro caso mostraremos un .

```

n = ARGV[0].to_i # 24
n.times do |i|
  if i%4 == 0 || i%4 == 1
    print '*'
  else
    print "."
  end
end
end

```

```

** ** ** ** **
.. .. .. ..

```

Desafío 5

Escribir el programa `patron3.rb` que permita dibujar el siguiente patrón:

```

..** | ..** | ..** |

```

Ejemplo de uso

```

ruby patron3.rb 4

```

```

..** | ..

```

Capítulo: Introducción a ciclos anidados

Objetivos

- Utilizar ciclos anidados para resolver problemas
- Conocer el concepto de complejidad

Introducción

Un ciclo anidado es simplemente un ciclo dentro de otro ciclo.

No existe ningún límite respecto a cuántos ciclos pueden haber anidados dentro de un código, aunque por cada uno aumentará la **complejidad temporal** del programa.

Los ciclos anidados expanden el universo de los tipos de problemas que podemos resolver.

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?

Fácil, envolviendo el código anterior en otro ciclo que itere de 1 a 10.

Tablas de todos los números

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

```
0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
0 * 3 = 0
0 * 4 = 0
0 * 5 = 0
0 * 6 = 0
0 * 7 = 0
0 * 8 = 0
0 * 9 = 0
1 * 0 = 0
1 * 1 = 1
...
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
```

Complejidad de un programa

Los ciclos anidados son una buena herramienta cuando es necesario usarlos, pero hay que tener cuidado con exagerar.

Cuando utilizamos un ciclo para recorrer un conjunto de datos, la cantidad de iteraciones depende directamente de la cantidad de datos que se recorren.

Cuando se ocupan ciclos anidados para revisar conjuntos de datos, lo que está adentro de ambos ciclos se ejecutará una cantidad de veces mucho mayor. Si es posible hacer la misma tarea usando un ciclo después de otro, o incluso un solo ciclo, el tiempo que se demorará el programa será considerablemente menor. Esto es más notorio para conjuntos más grandes de datos.

Por ejemplo, si tenemos un conjunto de 200 datos y otro de 50, y hacemos ciclos anidados recorriendo ambos, tendríamos ¡10 000 iteraciones! Por otro lado, si dejamos un ciclo después de otro, tendremos sólo 250 iteraciones en total.

Capítulo: Dibujando con ciclos anidados

Objetivos

- Reconocer patrones que requieran de más de un ciclo para resolverlos
- Utilizar ciclos anidados para dibujar patrones

Introducción

Los ejercicios de dibujo son ideales para practicar ciclos y especialmente ciclos anidados. ¿Qué tipo de cosas podemos dibujar con ciclos anidados?

Veamos el siguiente patrón:

```
*****
*****
*****
*****
*****
```

Solución

```
n = 5
# n = ARGV[0]

n.times do |i|
  n.times do |j|
    print '*'
  end
  print "\n"
end
```

Ejercicio medio triángulo

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

Ejemplo:

```
ruby medio_triangulo.rb 5
```

```
*  
**  
***  
****  
*****
```

Para resolver el código tenemos que observar el patrón:

- Si el usuario ingresa 5, se dibujan 5 filas.
- En la fila 1, hay un *
- En la fila 2, hay 2 *
- Por lo que podemos decir que en la fila n y hay n *

Las n filas

Primer acercamiento a la solución con n filas

```
n = ARGV[0] # 6  
  
n.times do |i|  
  puts '*'  
end
```

```
*  
*  
*  
*  
*  
*
```

Las columnas

```

n = ARGV[0]

n.times do |i|
  # Cuando i es 1 repetimos 1 vez
  # Cuando i es 2 repetimos 2 veces
  # Cuando i es N repetimos N veces
  # O sea que siempre estamos repitiendo i veces
  i.times do |j|
    print '*'
  end
  print "\n"
end

```

```

*
**
***
****
*****

```

Ejercicio triángulo

```

*
**
***
****
*****
*****
****
***
**
*

```

Ya sabemos cómo hacer la primera mitad

Nos concentraremos en la segunda parte de triángulo, donde el patrón que tenemos que armar es:

```

*****
****
***
**
*

```

Si n es igual a 5 dibujaremos 5 asteriscos, luego 4, luego 3, o iremos decrementando de uno en uno. Esto es lo mismo que decir `n - i`

- Iteración 0:
 - $5 - 0 = 5$
- Iteración 1:
 - $5 - 1 = 4$
- Iteración 2:
 - $5 - 2 = 3$
- Iteración n :
 - $5 - n$

```
n = ARGV[0] # 5

n.times do |i|
  (n-i).times do |j|
    print '*'
  end
  print "\n"
end
```

```
*****
****
***
**
*
```

Para armar el triángulo completo solo necesitamos juntar ambas soluciones:

```
n = ARGV[0] # 6

n.times do |i|
  i.times do |j|
    print '*'
  end
  print "\n"
end

n.times do |i|
  (n - i).times do |j|
    print '*'
  end
  print "\n"
end
```

```
*
**
***
****
*****
*****
*****
****
***
**
*
```

Ejercicio del cuadrado vacío

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

USO:

```
cuadrado_hueco.rb 3
```

```
resultado:
***
* *
***
```

```
cuadrado_hueco.rb 5
```

```
resultado:
*****
* *
* *
* *
*****
```

pista: Identifica cuál es la parte que se repite.

Solución al cuadrado vacío

La parte superior e inferior son similares y siempre son dos. La parte del medio, sin embargo, contempla el resto del cuadrado.

Primero dibujaremos la parte superior e inferior.

```

n = 5
#n = ARGV[0]

# Parte superior
n.times do |i|
  print "*"
end
print "\n"

# Parte inferior
n.times do |i|
  print "*"
end

```

La parte del medio consta siempre de dos asteriscos, uno al principio y el otro al final. El resto son espacios en blanco.

- Si n es 4, hay 2 asteriscos y 2 espacios.
- Si n es 5, hay 2 asteriscos y 3 espacios.
- Si n es 6, hay 2 asteriscos y 4 espacios.
- Si n es 7, hay 2 asteriscos y 5 espacios.

La cantidad de espacios siempre es `n - 2`

```

# n = 5

print "*"
(n - 2).times do |i|
  print " "
end
print "*"

```

Luego tenemos que repetir lo mismo todo n - 2 veces.

```

# n = 5
(n - 2).times do
  print "*"
  (n - 2).times do |i|
    print " "
  end
  print "*"
  print "\n"
end

```

```
* *  
* *  
* *
```

Finalmente unimos todo el código agregando las tapas.

```
n = ARGV[0].to_i  
  
# Parte superior  
n.times do  
  print "*"   
end  
print "\n"  
  
# Parte del medio  
(n - 2).times do  
  print "*"   
  (n - 2).times do  
    print " "   
  end  
  print "*"   
  print "\n"  
end  
  
# Parte inferior  
n.times do  
  print "*"   
end
```

Ejercicio resuelto

Se pide crear el programa `listas_y_sublistas.rb` donde el usuario ingrese un número como argumento y se genere una lista de HTML con esa cantidad de ítems y un segundo número que indique la cantidad de sub ítems.

Uso:

```
ruby listas_y_sublistas.rb 3 2
```

```
<ul>  
  <li>  
    <ul>  
      <li> 1.1 </li>  
      <li> 1.2 </li>
```

```
</ul>
</li>
<li>
  <ul>
    <li> 2.1 </li>
    <li> 2.2 </li>
  </ul>
</li>
<li>
  <ul>
    <li> 3.1 </li>
    <li> 3.2 </li>
  </ul>
</li>
</ul>
```

Pistas:

- Puedes tabular con `"\t"`
- Puedes hacer un salto de línea con `"\n"`

Solución

Lo primero que tenemos que hacer es identificar las partes que se repiten de las que no. Dentro de cada lista hay un patrón que se repite

```
<li> 1.1 </li>
<li> 1.2 </li>
```

Pero además, fuera de la lista, hay otro patrón que se repite:

```
<li>
  <ul>
    <li> 1.1 </li>
    <li> 1.2 </li>
  </ul>
</li>
```

Así que ocuparemos 2 iteraciones: Una para el ciclo interior y otra para el ciclo exterior.

Hacemos la lista interna primero:


```

n_externo = ARGV[0]
n_interno = ARGV[1]

n_interno.times do |i|
  puts "<li> #{i} </li>"
end

```

El siguiente paso es agregar el ciclo externo.

```

n_externo = ARGV[0]
n_interno = ARGV[1]

n_externo.times do |j|
  puts "<li>\n"
  puts "\t<ul>"
  n_interno.times do |i|
    puts "\t\t<li> #{j}.#{i} </li>"
  end
  puts "\t</ul>"
  puts "</li>"
end

```

Finalmente agregamos las etiquetas ` ` al principio y al final respectivamente.

```

n_externo = ARGV[0]
n_interno = ARGV[1]

puts "<ul>"
n_externo.times do |j|
  puts "<li>\n"
  puts "\t<ul>"
  n_interno.times do |i|
    puts "\t\t<li> #{j}.#{i} </li>"
  end
  puts "\t</ul>"
  puts "</li>"
end
puts "</ul>"

```

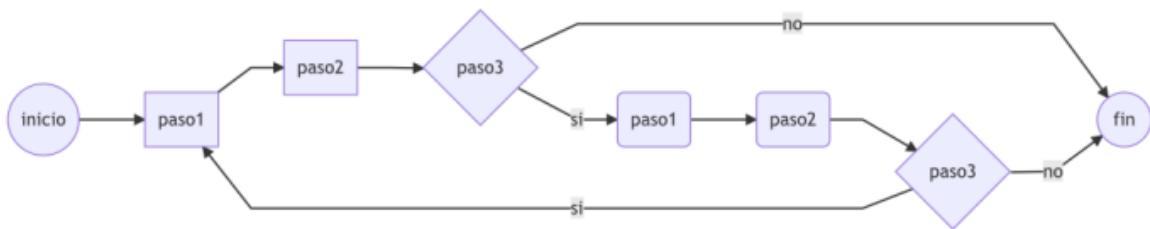
Capítulo: Reutilizando código

Objetivos

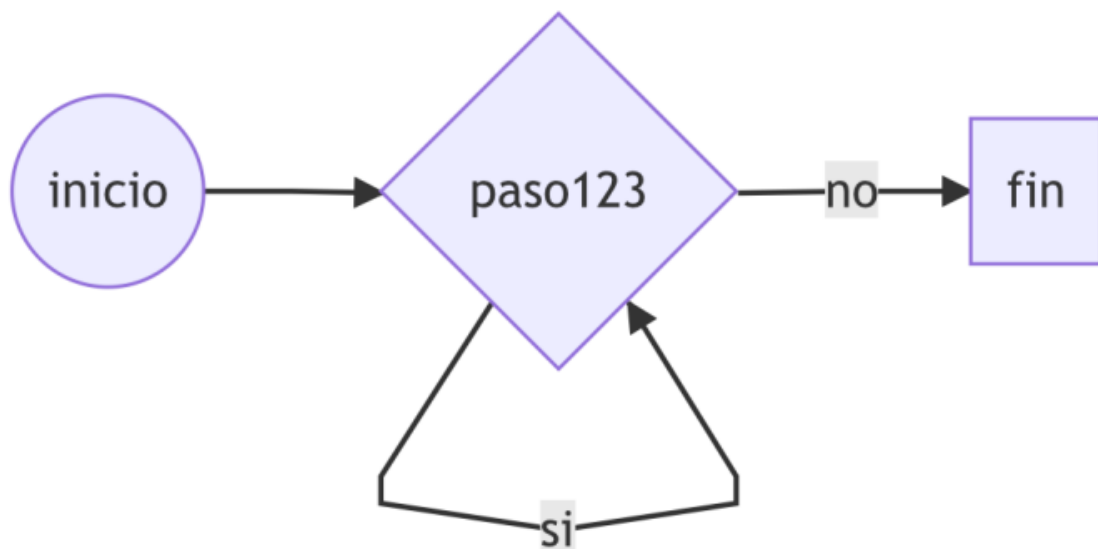
- Conocer la importancia de los métodos.
- Diferenciar entre la definición de un método y su llamado.

Motivación

Los métodos nos permiten abstraernos del cómo resolvemos los problemas, esto lo logramos agrupando instrucciones bajo un nombre. Supongamos que tenemos el siguiente algoritmo:



¿Qué pasaría si renombramos el paso1 y el paso2 y el paso3 a PasoX?, ¿sería lo mismo?



Diferentes nombres

Esta forma de agrupar instrucciones recibe diferentes nombres según el lenguaje de programación:

- Procedimiento

- Subrutina
- Función
- Método

El mismo propósito

Pero todas estas tienen el mismo propósito: Evitar que el programador repita (o copie y pegue) código, esto se llama enfoque **DRY** (Do not Repeat Yourself)

No son exactamente lo mismo

Si bien existen diferencias entre procedimiento, subrutina, función y método; por ahora las ignoraremos, ya que en Ruby sólo trabajaremos con métodos.

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**.

Llamar vs Definir

- Cuando creamos un método diremos que lo estamos definiendo.
- Cuando ocupemos un método diremos que lo estamos llamando, usando o invocando.

Dentro de Ruby hay varios métodos definidos que hemos utilizado, por ejemplo:

- `gets`
- `puts`

Definiendo métodos

Existen distintos lugares donde se puede definir métodos y eso tiene consecuencias sobre como los llamamos.

- El espacio principal de trabajo, ejemplo: `gets` y `puts` y `rand`.
- Dentro de objetos, ejemplo: `2.even?`
- Dentro de clases, ejemplo: `File.open`
- Dentro de módulos, ejemplo: `Math.sqrt`

El espacio principal (main)

En esta unidad trabajaremos definiendo los métodos en el espacio principal.

Sabemos que un método está definido dentro del espacio principal cuando lo ocupamos directamente, sin anteponer un objeto, clase o módulo.

Los métodos suelen organizarse entorno a clases y módulos, y podemos cargarlas desde un archivo o desde **gemas**. En los próximos capítulos aprenderemos a crear nuestros propios métodos.

Capítulo: Creando métodos

Objetivos

- Definir métodos.
- Llamar métodos.
- Conocer el orden en que se ejecuta un código que posee métodos.

Introducción

En este capítulo aprenderemos a crear métodos desde cero. Esto nos permitirá simplificar códigos que ya hemos escrito y reutilizar código.

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.
```

Todas las instrucciones definidas dentro del método serán ejecutadas cuando hagamos un llamado a este.

Creando nuestro primer método

```
def imprimir_menu  
  puts 'Menú: '  
  puts '1) Opción 1'  
  puts '2) Opción 2'  
  puts '3) Opción 3'  
  puts '4) Salir'  
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()
```

Los paréntesis son optativos

Los paréntesis son optativos, sin embargo, la guía de estilo recomienda no utilizarlos.

Código más limpio es mejor código :)

Cuidado con los paréntesis

Al utilizar paréntesis estos tienen que estar junto al nombre u obtendremos un error.

```
imprimir_menu () #ArgumentError: wrong number of arguments (given 1, expected 0)
```

Este error lo estudiaremos en el siguiente capítulo cuando estudiemos qué es un argumento.

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
```

Esto es equivalente a ocupar una variable antes de asignarle un valor.

Pero cuando nosotros inspeccionamos un código, es más sencillo comenzar leyendo desde el punto de partida.

```
def imprimir_menu  
  puts 'Escoge una opción:'  
  puts '-----'
```

```
puts '1 - Acción 1'
puts '2 - Acción 2'
puts 'Escribe "salir" para terminar el programa'
puts 'Ingresa una opción:'
end

# Partimos leyendo desde aquí.
opcion_menu = 'cualquier valor'
while opcion_menu != 'salir' || opcion_menu != 'Salir'
  imprimir_menu # Aquí leemos imprimir_menu
  opcion_menu = gets.chomp
  if opcion_menu == '1'
    puts 'Realizando acción 1'
  elsif opcion_menu == '2'
    puts 'Realizando acción 2'
  elsif opcion_menu != 'salir' || opcion_menu != 'Salir'
    puts 'Saliendo'
  else
    puts 'Opción inválida'
  end
end
```

Capítulo: Parametrizando métodos

Objetivos

- Crear métodos que reciben parámetros.
- Diferenciar parámetros de argumentos.
- Crear métodos con parámetros opcionales.

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
```

Llamando a un método que exige parámetros

```
incrementar(5)
# => El resultado es 6

incrementar 10 # Los paréntesis son optativos
# => El resultado es 11

incrementar 51
# => El resultado es 52
```

Decimos que los parámetros se exigen porque si no los especificamos, obtendremos un error:

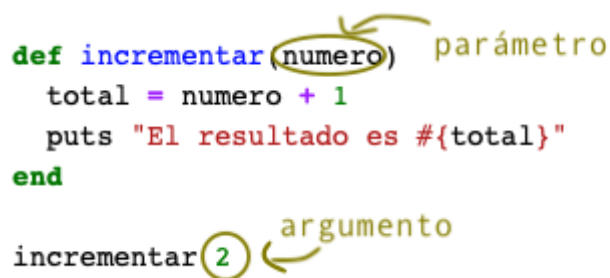

```
def incrementar(numero)
  total = numero + 1
  puts "El resultado es #{total}"
end

incrementar #Wrong number of arguments (given 0, expected 1)

# Tampoco podemos pasar argumentos de más
incrementar 2,4 #Wrong number of arguments (given 2, expected 1)
```

¿Qué es un argumento?

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



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

incrementar 2
```

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

Otra forma de verlo es decir que parámetro es un valor genérico, no sabemos exactamente cuál es el valor que se va a pasar. En cambio, hablamos de argumento cuando especificamos el valor.

Cuidado con los paréntesis

Tenemos que tener cuidado al utilizar espacios y paréntesis

```
incrementar 2 # ruby lo lee como incrementar(2)
incrementar (2) #ruby lo lee como incrementar((2)), esto no es un problema
```

Pero si el método no recibe parámetros

```
def parentesis
  puts "x"
end

parentesis () # ruby lo lee como parentesis(())
```

parentesis(()) lo que es lo mismo que parentesis(nil), o sea estamos pasando un parámetro y el método no recibe parámetros por lo que obtendremos el error:

```
# (wrong number of arguments (given 1, expected 0))
```

Una variable puede ser usada como argumento

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

a = 2
incrementar(a)
```

Esto lo utilizaremos con bastante frecuencia.

Un método puede recibir más de un parámetro

Podemos especificar todos los parámetros que queramos, simplemente separándolos por una coma.

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

incrementar(2, 3)
incrementar 3, 3 # El uso de paréntesis es opcional
```

Si el método exige dos parámetros tenemos que pasarle 2 argumentos

```
incrementar 2 # ArgumentError: wrong number of arguments (given 1, expected 2)
```

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

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

Un valor opcional
tiene un valor asignado
por defecto



En este caso sólo uno de los valores es opcional (pero podríamos crear un método donde todos los parámetros fueran opcionales).

Si no le pasamos ningún argumento veremos el siguiente error:

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

incrementar
# ArgumentError: wrong number of arguments (given 0, expected 1..2)
```

1..2 se lee como que espera **mínimo 1** y **máximo 2**.

Los métodos deben ser reutilizables

Veamos el mismo código escrito de dos formas distintas.

```
# Opción 1:
def fahrenheit()
  puts 'Ingresa la temperatura en fahrenheit'
  fahrenheit = gets.to_i
  celsius = (fahrenheit + 40) / 1.8 - 40
  puts "la temperatura es de #{celsius} celsius"
end
```

```
# Opción 2:
def fahrenheit(f)
  celsius = (f + 40) / 1.8 - 40
  puts "la temperatura es de #{celsius} celsius"
end

puts 'Ingresa la temperatura en fahrenheit'
fahrenheit(gets.to_i)
```

¿Cuál es más flexible? Esto es lo que discutiremos a continuación.

Ejemplo de reutilización

¿Qué pasaría si ya no queremos que el usuario ingrese los valores, y queremos generar los valores al azar? El segundo es más flexible porque nos permite llamarlo independiente de cómo se ingresen los valores.

```
# Opción 2:
def fahrenheit(f)
  celsius = (f + 40) / 1.8 - 40
  puts "la temperatura es de #{celsius} celsius"
end

puts 'Ingrese la temperatura en fahrenheit'
fahrenheit(rand(50))
```

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

```
def validar_edad
  edad = gets.to_i
  if edad >= 18
    puts "es mayor"
  else
    puts "es menor"
  end
end
```

Resumen

- Los parámetros nos permiten hacer los métodos muy flexibles.
- Algunos parámetros pueden ser opcionales, para eso debemos asignarles un valor por defecto.

```
def prueba(x = 2)
  puts x
end

prueba
prueba(3)
prueba 3
```

- Las variables pueden ser utilizadas como argumentos.

```
def prueba(x = 2)
  puts x
end

a = 5
prueba(a)
```

Capítulo: Retorno

Objetivo

- Entender la importancia del retorno de un método.
- Crear métodos con retorno.
- Saber que los retornos en Ruby son, por defecto, implícitos.

Motivación

En muchas situaciones necesitaremos utilizar el resultado de un método para seguir trabajando con él.

Es escasas ocasiones los métodos muestran información directo en pantalla, normalmente retornan el valor para que puedas seguir trabajando.

También será muy frecuente que, en nuestro trabajo, nos soliciten un método que reciba cierta información, la procese y devuelva algo.

Creando nuestro primer método con retorno

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

De esta forma si queremos mostrar el resultado simplemente escribiremos:

```
puts transformar_a_fahrenheit(123)
```

Al separar el imprimir del método lo hacemos mucho más flexible.

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

puts transformar_a_fahrenheit(123)
```

1) f = 123

2) 50.556

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
```

El retorno implícito es siempre sobre la última línea

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
```

Recordemos que `nil` es la ausencia de un valor.

¿Implícito o Explícito?

La instrucción `return` no se usa frecuentemente en Ruby. En la mayoría de los casos basta con un retorno implícito.

Cuidado con los puts y prints

En los métodos el retorno es implícito, y esto quiere decir que el `return` se hace sobre la última línea.

```
def fahrenheit(f)
  celsius = (f + 40) / 1.8 - 40
  puts celsius # puts celsius devuelve nil
end

# fahrenheit(45) ==> nil
```

La instrucción `puts` siempre retorna `nil`. Es por esto que debemos tener cuidado con utilizar la instrucción `puts` en la última línea de un método.

Resumen

- Retorno, devolver, salida son sinónimos para el valor que devuelve un método.
- Podemos hacer que un método devuelva un valor, de manera explícita, utilizando la instrucción `return`.
- La última línea de un método tiene un `return` implícito.
- Retornar **NO ES LO MISMO** que mostrar en pantalla.
- `puts` devuelve `nil`, por lo que agregar un `puts` en la última línea de un método hace que este devuelva `nil`.
- `nil` es un objeto que representa la ausencia de un valor.

Capítulo: Alcance de variables

Objetivos

- Conocer los tipos de variable.
- Conocer el concepto de alcance.
- Diferenciar variables locales de variables globales.
- Entender qué sucede con las variables dentro de un método.

Motivación

Los tipos de variable y el alcance de estas son conceptos muy importantes, nos permiten entender desde dónde podemos acceder a una variable.

En este capítulo estudiaremos las reglas de alcance de las variables locales y globales.

Tipos de variable

En Ruby existen 4 tipos de variable:

- Globales
- Locales
- De instancia
- De clase

Las últimas dos se ocupan dentro de la creación de objetos, por lo que no las abordaremos todavía.

Nos enfocaremos en las variables globales y locales.

El alcance

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

Alcance de una variable local

Una variable **definida dentro de un método no puede ser accedida fuera del método**. A las reglas desde dónde puede ser accedida a una variable se le denomina alcance.

```
def aprobado?(nota1, nota2)
  promedio = (nota1 + nota2) / 2
  promedio >= 5 ? true : false
end

aprobado?(4, 5) # false
aprobado?(10, 5) # true
```

```
true
```

```
puts promedio # undefined local variable or method `promedio'
```

Los parámetros también cuentan como variables locales

```
def aprobado?(nota1, nota2)
  promedio = (nota1 + nota2) / 2
  promedio >= 5 ? true : false
end

aprobado?(4, 5) # false
puts nota1 # undefined local variable or method `nota1' for main:Object
```

No importa el orden de declaración

Es un tema de espacio de trabajos, no de orden.

```
nombre = 'Montgomery Burns'

def saludar
  puts "Hola #{nombre}!"
end

# undefined local variable or method `nombre'
```

Main

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

# Esto vuelve a ser el ambiente 'main'
# Aquí no existe las variables palabra ni numero
puts nombre
puts edad

```

Cuando ingresamos en un método, estamos trabajando en un ambiente nuevo que, aunque posea variables con el mismo nombre, **no afecta las variables 'externas'**.

```

# El alcance de estas variables es 'main'
name = 'Homero Simpson'
age = 40
address = 'Springfield'
human = false

def presentar
  # El alcance de las variables definidas aquí es el método
  # Estas instrucciones NO afectan las variables de 'main'
  name = 'Milhouse Van Houten'
  age = 10
  occupation = 'student'

  puts "#{name} tiene #{age} años!"
end

presentar

puts "#{name} tiene #{age} años!"

```

Este tipo de variable se conoce como **variables locales**, por consiguiente, podemos afirmar que:

- `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`.

Existe un método para saber si una variable es local

```
defined? edad
# => "local-variable"
```

Variables globales

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

En Ruby las variables globales se definen comenzando por un `$`. Ingreseemos a IRB y hagamos la siguiente prueba:

```
$continente = 'Sudamérica'
# => "Sudamérica"

defined? $continente
#=> "global-variable"
```

Al estar definida como una variable global, podemos operar con ellas dentro y fuera de nuestros métodos

Veámoslo en la práctica:

```
# Definición de la variable global
$continente = 'Sudamérica'

def modificar_continente(nombre_continente)
  # La variable global si existe dentro del método
  $continente = nombre_continente
end

# Llamado al método que modifica el nombre de la variable global
modificar_continente('Europa')

puts $continente
# "Europa"
```

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?

Un programa puede tener miles de líneas de código e incorporar varias bibliotecas (programas de tercero). 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.

Existen muchas otras razones por las que estas no se recomiendan, pero que las dejaremos, por ahora, fuera de discusión.

Cierre

En esta unidad aprendimos sobre:

- Ciclos
- Métodos.

Ciclos

Los ciclos nos ayudan a resolver problemas en base a iteraciones y son muy esenciales para la resolución de problemas.

- Algunos tipos de ciclos como `.times` pueden recibir bloques
- Las variables (locales) definidas dentro de los bloques no pueden ser accedidas desde fuera del bloque.
- Al utilizar `while` tenemos que tener cuidado de cambiar el índice para que haya un punto de salida del ciclo.

Métodos

Los métodos nos permiten reutilizar código. Sobre métodos aprendimos:

- Pueden recibir parámetros
- El retorno de la última línea es implícito
- Las variables definidas dentro de los métodos no pueden ser accedidas desde fuera