

## Lectura complementaria

### ¿Se puede 'romper' un ciclo?

La respuesta es sí.

Existen instrucciones que nos permiten tanto, saltar a la siguiente iteración, como salir completamente de un ciclo.

#### La instrucción *break*

Se utiliza, como su nombre lo indica, para 'romper' un ciclo; en otras palabras, para salir completamente de la iteración.

La instrucción `break` se puede utilizar de manera directa aunque, en la práctica, suele ir acompañada de una condición.

```
10.times do |i|  
  puts i  
  break  
end  
  
# 0
```

```
0
```

En el ejemplo:

- Comienza la iteración
- Imprime valor de `i = 0`
- Sale del ciclo

Observemos, a continuación, la instrucción acompañada de una condición:

```
i = 0  
while i < 100
```

```
puts i
if i > 5
  break
end
i = i + 1
end
```

```
# 0
# 1
# 2
# 3
# 4
# 5
# 6
# => nil
```

```
0
1
2
3
4
5
6
```

¿Qué debemos modificar si queremos que, además de salir del ciclo, no imprima valores mayores a 5? Solucionemos ese problema y hagamos un pequeño refactor:

```
i = 0
while (i < 100)
  break if i > 5 # if inline
  puts i # Ahora no imprime valores mayores a 5 ya que está después de break
  i += 1 # Contador
end
```

```
0
1
2
3
4
5
```


## La instrucción *next*

La instrucción `next` nos permite, usualmente acompañada de una condición, 'saltar' las instrucciones de la iteración actual.

Por ejemplo, queremos imprimir sólo los números impares de una iteración. Una de las posibles soluciones es saltar a la siguiente iteración si el número actual es par.

¿Cómo podemos identificar si un número es par?

Hint: Revisemos la documentación en busca de un método que nos permita evaluar si un entero es par:

 **even? → true or false**  
Returns `true` if `int` is an even number.

```
[2.5.1 :001 > 2.even?  
=> true  
[2.5.1 :002 > 3.even?  
=> false
```

Apliquemos ahora este método y la instrucción `next`

```
20.times do |i|  
  next if i.even?  
  puts i  
end
```

```
1  
3  
5  
7  
9  
11  
13  
15  
17  
19
```

```
20
```

Recordemos que, en programación, existen diversas formas de dar solución a un problema. Debemos privilegiar las más sencillas.

Probablemente una solución más sencilla al último ejemplo es condicionar directamente la impresión:

Hint: Consultar método `odd?` en la documentación.

```
20.times do |i|  
  puts i if i.odd?  
end
```

```
1  
3  
5  
7  
9  
11  
13  
15  
17  
19  
  
20
```

Sin embargo, en escenarios más complejos, la instrucción `next` puede ser bastante útil.

## La instrucción Until

La instrucción `until` no es más que la versión negativa de `while`:

- `while` ejecuta código **mientras** se cumpla una condición.
- `until` ejecuta código **hasta que** se cumpla una condición

Podemos escribir el ejemplo anterior utilizando la `until`

```
i = 0  
until i == 10  
  puts "Esto se imprimirá 10 veces"  
  i += 1  
end
```

Ambas instrucciones `while` y `until` tienen, al igual que `if`, una variante *inline*.

¿Qué hace el siguiente bloque de código?

```
puts "Ingrese su password: "  
password = gets.chomp until password == 'secreto'
```

# Capítulo: Ciclos anidados

---

## Objetivos

- Leer y transcribir diagramas de flujo con iteraciones anidadas a código Ruby
- Crear un menú con submenú en Rub

Es perfectamente posible anidar ciclos, es decir, realizar una iteración dentro de otra.

Desarrollemos un programa que nos permita imprimir la tabla de multiplicar del 1 al 9 utilizando la el ciclo `times` :

```
10.times do |i|
  next if i == 0
  10.times do |j|
    next if j == 0
    puts "#{i} * #{j} = #{i*j}"
  end
end
```

## Ejercicio de integración

Construyamos un menú con submenú. Se solicita:

- Se debe imprimir un menú con opciones.
- Usuario debe ingresar una opción.
- Si se ingresa una opción entre 1 y 3: Debe mostrar submenú correspondiente
- El submenú muestra 4 opciones. Por ejemplo si del menú se eligió la opción 2:
  - 2.1) Opción 1
  - 2.2) Opción 2
  - 2.3) Opción 3
  - 2.4) Volver al menú principal
- Al seleccionar un opción del submenú, esta debe ser mostrada en pantalla. Luego volver al submenú.
  - "Se ha seleccionado la opción 2.3"
- Se debe validar que la opción ingresada, tanto en menú como submenú, sea válida.
- Si en el menú principal se ingresa la opción 4, el programa termina.

**Intenta resolver este ejercicio por tu cuenta, guíate por lo siguiente pasos:**

1. Analizar el problema
2. Descomponer el problema

3. Imaginar posibles soluciones
4. Realizar un diagrama de flujo
5. Validar el diagrama de flujo
6. Programar la solución

```
puts 'Menú: '  
puts '1) Opción 1'  
puts '2) Opción 2'  
puts '3) Opción 3'  
puts '4) Salir'  
  
puts 'Ingrese una opción: '  
opcion_menu = gets.chomp.to_i  
  
while opcion_menu != 4  
  if opcion_menu < 1 || opcion_menu > 4  
    puts 'Opción no es válida'  
  else  
    puts 'Submenú: '  
    puts "#{opcion_menu}.1) Opción 1"  
    puts "#{opcion_menu}.2) Opción 2"  
    puts "#{opcion_menu}.3) Opción 3"  
    puts "#{opcion_menu}.4) Volver al menú principal"  
  
    puts 'Ingrese una opción: '  
    opcion_submenu = gets.chomp.to_i  
  
    while opcion_submenu != 4  
      if opcion_submenu < 1 || opcion_submenu > 4  
        puts 'Opción no es válida'  
      else  
        puts "La opción ingresada fue #{opcion_menu}.#{opcion_submenu}"  
        puts 'Submenú: '  
        puts "#{opcion_menu}.1) Opción 1"  
        puts "#{opcion_menu}.2) Opción 2"  
        puts "#{opcion_menu}.3) Opción 3"  
        puts "#{opcion_menu}.4) Volver al menú principal"  
      end  
      puts 'Ingrese una opción: '  
      opcion_submenu = gets.chomp.to_i  
    end  
  end  
end  
  
puts 'Menú: '  
puts '1) Opción 1'  
puts '2) Opción 2'  
puts '3) Opción 3'  
puts '4) Salir'  
  
puts 'Ingrese una opción: '  
opcion_menu = gets.chomp.to_i  
end
```

El programa funciona, sin embargo, podemos observar que existen varias líneas de código (como la impresión del menú y submenú) que se repiten. En programación, la repetición de líneas de código suele ser una mala práctica. Para evidenciar esta mala práctica y tratar de corregirla se hace presente la filosofía del código **DRY**.

El principio **No te repitas** (en inglés Don't Repeat Yourself o **DRY**) es una filosofía promueve la reducción de la duplicación de código en nuestros programas. Según este principio, nuestro código, nunca debería ser duplicado debido a que esto dificulta eventuales cambios, puede perjudicar la claridad y escalabilidad un espacio para posibles inconsistencias.

Existen muchas herramientas, patrones de diseño y buenas prácticas que nos ayudan a promover este principio. Una de estas herramientas corresponde a la creación y utilización de métodos propios, concepto que estudiaremos en la siguiente unidad.

# Capítulo: Analizando métodos

---

## Objetivo

- Mejorar la calidad de código de nuestros métodos
- Conocer la convención de los métodos con `?`

## Introducción

En este capítulo vamos a discutir algunos usos típicos de métodos y como escribirlos con menos código y mas claridad.

```
def mayor_de_edad?(edad)
  true if edad >= 18
end

mayor_de_edad?(20)
```

Al igual que cuando analizamos casos de borde en la unidad anterior.

## Si la edad es menor a 18 ¿Qué retornará nuestro método?

El comportamiento esperado sería `false`, sin embargo, no hemos definido ese flujo y nuestro método retornará `nil`.

Vamos a corregir nuestro método:

```
def mayor_de_edad?(edad)
  if edad >= 18
    true
  else
    false
  end
end

mayor_de_edad?(20) # true
mayor_de_edad?(10) # false
```

Si el argumento es mayor o igual a 18, entonces la condición se evalúa como verdadera y el flujo ingresa a `true`. Luego sale del `if`, convirtiendo a `true` en la última instrucción evaluada en el método. Lo mismo para el flujo `false`.



Es una convención que los métodos que devuelven `true` o `false` se definan finalizando su nombre con un signo de interrogación. Ya hemos utilizado este tipo de sintaxis: `.even?` o `.nil?`

## ¿Y ahora qué retorna?

```
def mayor_de_edad?(edad)
  if edad >= 18
    true
  else
    false
  end
  puts edad
end
```

## ¿Y ahora?

```
def mayor_de_edad?(edad)
  edad >= 18
end
```

## ¿Qué retorna el siguiente método?

```
def trabalenguas_uno
  "Tres tristes tigres tragaban trigo en un trigal en tres tristes trastos"
end
```

## ¿Qué se imprime en pantalla?

```
def trabalenguas_uno
  "Tres tristes tigres tragaban trigo en un trigal en tres tristes trastos"
end

def trabalenguas_dos
  puts "El rey de Constantinopla está constantinopolizado"
end

puts trabalenguas_uno
puts trabalenguas_dos
```

Ambas soluciones son aceptadas, el uso de una u otra dependerá del escenario en el que nos encontremos.

- El método `trabalenguas_uno` retorna un string con el que, además de imprimir, podemos operar.
- El método `trabalenguas_dos` imprime y luego retorna `nil`.

## ¿Qué se imprime pantalla?

```
def trabalenguas_uno
  "Tres tristes tigres tragaban trigo en un trigal en tres tristes trastos"
end

def trabalenguas_dos
  puts "El rey de Constantinopla está constantinopolizado"
end

puts trabalenguas_uno.length
puts trabalenguas_dos.nil?
```

```
# 71
# El rey de Constantinopla está constantinopolizado
# true
```

No importa que operación apliquemos al retorno del método `trabalenguas_dos`, simplemente la acción de llamar al método implica la impresión del String.

## El retorno implícito es el retorno más utilizado en Ruby

¿Eso quiere decir que existe otro tipo de retorno?

La respuesta es sí. Podemos forzar un retorno previo a la última línea utilizando la instrucción `return`.

```
def impresion
  puts 'Este método finaliza'
  return 'cuando encuentra un retorno'
  puts 'Sin importar'
  puts 'La cantidad de instrucciones'
end
```

Debido a la instrucción `return`, el método `impresion` retorna el String `'cuando encuentra un retorno'`.

## TIP Avanzado

En la mayoría de los casos bastará con un retorno implícito; sin embargo, en algunas ocasiones, la utilización de la instrucción `return` puede simplificar nuestro algoritmo.

El siguiente método realiza una serie de instrucciones sobre un archivo **a menos que** el argumento con el nombre del archivo sea un String vacío:

```
def save_to_file(filename)
  unless filename.empty?
    do_something
    do_something_else
    File.write #... blah blah
  end
end
```

La utilización de una instrucción `return` hace que el método retorne `false` en cuanto valide que el argumento es un String vacío:

```
def save_to_file(filename)
  return false if filename.empty? # << Cláusula de guardia
  do_something
  do_something_else
  File.write #... blah blah
end
```

Esta estructura recibe el nombre de cláusula de guardia.

## Profundizando en métodos

El siguiente método recibe como parámetro la suma de las notas de un alumno, calcula el promedio y luego retorna `true` si el alumno está aprobado o `false` si está reprobado. La nota de aprobación es 4:

```
def aprobado?(promedio)
  promedio >= 4
end
```

Podemos hacer que este método calcule el promedio con nota 4, a menos que, enviemos otro argumento correspondiente a la nota de aprobación:

```
def aprobado?(promedio, nota_de_aprobacion = 4)
  promedio >= nota_de_aprobacion
end
```

La acción de asignar un valor al parámetro se denomina **parámetro por defecto** y hace que nuestro método se vuelva flexible con respecto al número de argumentos a recibir.

- Si sólo enviamos el promedio, se utilizará como `nota_de_aprobacion` el valor `4`.
- Si enviamos un segundo argumento, la nota de aprobación será reemplazada por el valor del argumento.

```
def aprobado?(promedio, nota_de_aprobacion = 4)
  promedio >= nota_de_aprobacion
end
```

```
aprobado?(5) # true: El promedio es 5 y la nota de aprobación es 4.
```

```
aprobado?(5,6) # false: El promedio es 5 y la nota de aprobación es 6.
```

# Capítulo: Introducción a gemas

---

## Objetivos

- Conocer qué es un gestor de paquetes
- Conocer qué es una gema
- Conocer rubygems
- Byebug
- Conocer concepto de 'debugear' y su importancia en programación
- Instalar una gema en Ruby
- Utilizar byebug para debugear un script en Ruby
- Utilizar byebug para seguir el flujo de un programa en Ruby

## Bibliotecas en programación

En la mayoría de los lenguajes de programación existen herramientas prefabricadas por terceros que nos ayudan a simplificar la implementación de ciertas tareas.

Estas herramientas se denominan **libraries** o **bibliotecas**.

## No hay que reinventar la rueda

En programación, las bibliotecas son ampliamente utilizadas y están disponible tanto para tareas cotidianas como para procesos complejos.

La filosofía detrás del uso de bibliotecas en nuestros programas es que no necesitemos 'reinventar la rueda', sino más bien, aprovechemos una herramienta ya fabricada que cumpla con nuestros requerimientos.

## Gestor de paquetes

Sumado a lo anterior, la utilización de bibliotecas va de la mano con una **herramienta que nos ayude a gestionarlas**.

Esta herramienta se conoce comúnmente como **gestor de paquetes** y su función es -entre otras- simplificar las tareas de instalación, actualización, eliminación, etc. en lo que respecta a bibliotecas.

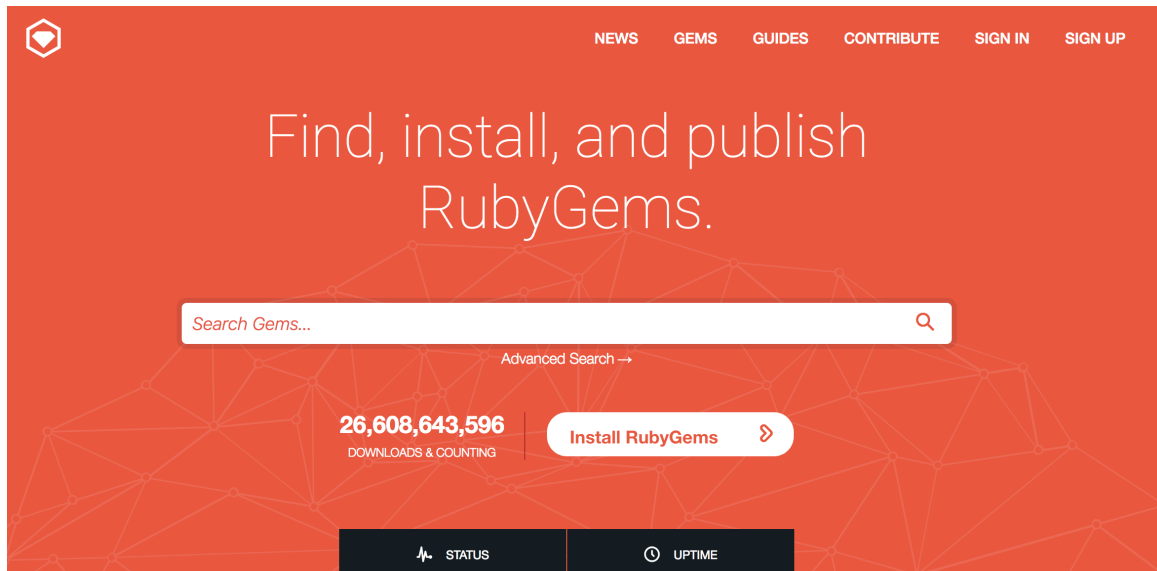
## Gemas

En Ruby, las bibliotecas reciben el nombre de **gemas**.

Una gema no es más que la forma en que las librerías de Ruby son empaquetadas. Dentro de toda gema encontraremos código Ruby (archivos `.rb`).

Como mencionamos con anterioridad, existen gemas para -casi- todo y la herramienta que nos ayuda a gestionar la implementación de gemas se denomina **RubyGems**.

RubyGems, además de funcionar como gestor de paquetes, nos ofrece un servidor para la distribución oficial de estos llamado [RubyGems.org](https://rubygems.org) donde podemos tanto buscar e instalar gemas de terceros como publicar gemas de nuestra autoría.



Ruby, a partir de su versión 1.9, incluye RubyGems. Por lo tanto, podemos ir directo a la instalación de nuestra primera gema.

## Byebug

Revisamos anteriormente en esta unidad el concepto de 'debugear' y tuvimos un primer acercamiento utilizando la instrucción `puts` para identificar comportamientos no esperados en nuestro programa.

Sin embargo, existe una gema ampliamente utilizada llamada [Byebug](https://github.com/byebug/byebug) que nos ofrece funcionalidades que nos facilitan el proceso de debugging.

# byebug 10.0.2

Byebug is a Ruby debugger. It's implemented using the TracePoint C API for execution control and the Debug Inspector C API for call stack navigation. The core component provides support that front-ends can build on. It provides breakpoint handling and bindings for stack frames among other things and it comes with an easy to use command line interface.

## VERSIONS:

**10.0.2** - March 30, 2018 (80 KB)  
**10.0.1** - March 21, 2018 (80 KB)  
**10.0.0** - January 26, 2018 (80 KB)  
**9.1.0** - August 22, 2017 (78 KB)  
**9.0.6** - September 30, 2016 (77.5 KB)

[Show all versions \(75 total\) →](#)

## DEVELOPMENT DEPENDENCIES (1):

**bundler** ~> 1.7

## AUTHORS:

David Rodriguez, Kent Sibilev, Mark Moseley

 **2,621**

TOTAL DOWNLOADS

**48,771,155**

FOR THIS VERSION

**4,182,694**

## GEMFILE:

`gem 'byebug', '~> 1.7`

## INSTALL:

`gem install byebug`

## LICENSE:

**BSD-2-CLAUSE**

REQUIRED RUBY VERSION:

También podemos consultar la documentación y el código fuente desde el [repositorio de GitHub](#) oficial de la gema.

## Instalando Byebug

Una de las formas de instalar una gema es a través del terminal utilizando el comando `gem`. Este comando es proporcionado por RubyGems.

Siguiendo la [documentación](#), abrimos una nueva terminal y ejecutamos el siguiente comando:

```
gem install byebug
```

```
jpc1 🍏 JPG in ~/Desktop/intro-progra-ruby
>> gem install byebug
Fetching: byebug-10.0.2.gem (100%)
Building native extensions. This could take a while...
Successfully installed byebug-10.0.2
1 gem installed
```

Una vez finalizada la instalación de Byebug, el siguiente paso es incluir la gema en nuestro archivo para poder utilizarla. Para incluir la gema agregamos la siguiente línea al inicio de nuestro archivo:

```
require 'byebug'
```

Con esa instrucción, básicamente, le estamos diciendo a Ruby que disponibilice para nosotros las funcionalidades de byebug en ese archivo.