



# Introducción a la programación

---

## Capítulo: Presentación del módulo

---

### Objetivos

- Conocer el alcance del módulo.
- Conocer los requisitos del módulo.
- Conocer herramientas que utilizaremos en el módulo.
- Conocer a quién está dirigido el módulo.

### Alcance del módulo

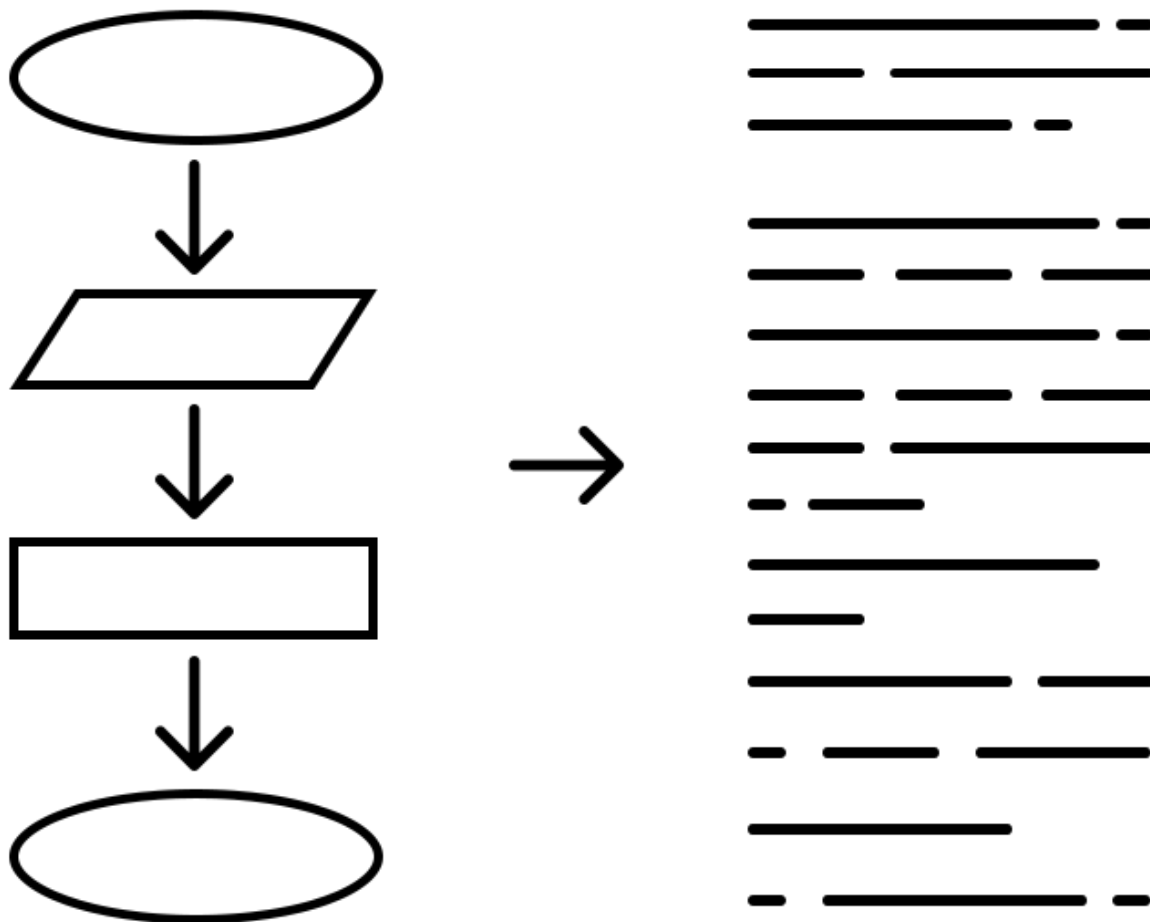
A lo largo del módulo aprenderemos a construir programas en Ruby que capturen información de fuentes de internet o de archivos, a procesar estos datos y guardar los resultados.

### Composición del módulo

Este módulo se compone de 4 unidades:

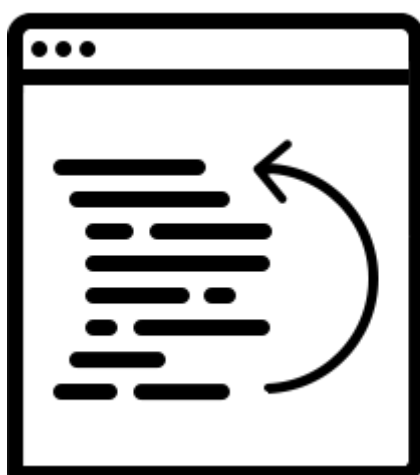
- Flujo
- Ciclos y métodos
- Arreglos y archivos
- Hashes y APIs

### Flujo



En la unidad de flujo aprenderemos a construir aplicaciones como una calculadora o una fórmula y programas que puedan tomar decisiones en base a valores.

## Ciclos y métodos



En esta unidad aprenderemos a construir aplicaciones que puedan generar y/o recorrer grandes cantidades de datos. Podremos construir programas que cuenten, resuelvan sumatorias o productorias o generen series de datos.

En esta unidad aprenderemos a identificar patrones, resolverlos ocupando ciclos y reutilizar código utilizando métodos.

## Arreglos y archivos

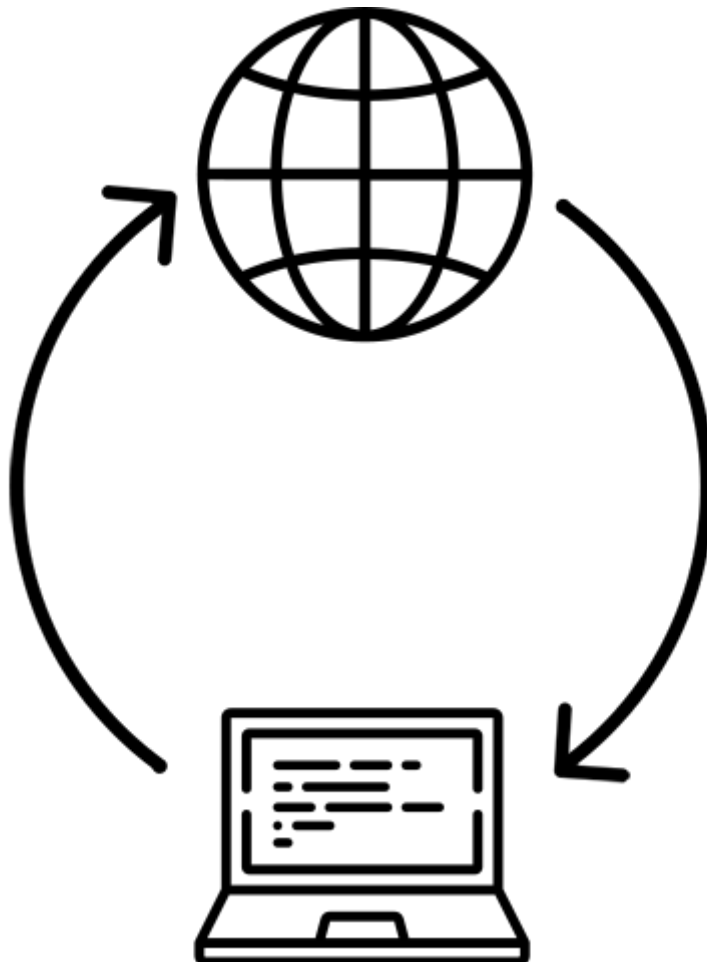
...	
—	20
—	50
— —	70
—	90

→ **[20, 50, 70, 90]**

En esta unidad aprenderemos a trabajar con grandes cantidades de datos que vienen entregados por el usuario al ejecutar el programa o desde un archivo.

Utilizando arreglos y ciclos aprenderemos a transformar y filtrar datos.

## Hashes y APIs



En esta unidad aprenderemos a obtener datos para nuestro programa desde fuentes en internet, estos datos los almacenaremos en arreglos y/o hashes y los procesaremos ocupando las técnicas aprendidas.

## ¿Para quién es este módulo?



Este módulo es una introducción a la programación utilizando Ruby desde un enfoque práctico. Está pensado para personas interesadas en aprender a programar y gente que tenga poca experiencia en programación y esté interesada en aprender un lenguaje nuevo.

Si tienes mucha experiencia programando pero no conoces Ruby, las dos primeras unidades se te harán muy familiares, pero las unidades 3 y 4 tienen varios elementos interesantes que son propios de Ruby y los necesitarás para trabajar en Rails.

## Requisitos para este módulo

- Manejo del terminal.
- Leer inglés técnico. Lo suficiente para entender los mensajes cuando cometamos errores.

## ¿Por qué tenemos que saber inglés?

```
puts c
Traceback (most recent call last):
  2: from /Users/gonzalosanchez/.rvm/rubies/ruby-2.5.3/bin/irb:11:in `'
  1: from (irb):7
NameError (undefined local variable or method `c' for main:Object)
Did you mean?  cb
```

Porque cuando estemos programando en algunas ocasiones cometeremos errores y nos toparemos con mensajes como el que acabamos de mostrar.

Si bien en este momento no tenemos que entender el error, tenemos que saber en términos muy generales de qué trata y poder buscarlo en Google.

## Terminal

De la terminal tenemos que saber:

- Entrar al programa.
- Conocer la estructura de directorios.
- Moverse en los directorios.
- Diferenciar una ruta absoluta de una relativa.
- Utilizar **git** y **github**.

## ¿Qué no se cubre en este módulo?

Este es un módulo de introducción a programación. No cubriremos programación orientada a objetos, desarrollo de aplicaciones web o manejo de bases de datos. Estos temas son importantes para un desarrollador, especialmente para un desarrollador Fullstack, por lo mismo los cubriremos en una próxima unidad cuando comprendamos las bases de la programación.

## Ordenando el material de estudio

Es importante tener ordenado nuestro material de estudio, principalmente para repasar. Es por esto mismo que crearemos todos los proyectos en una carpeta que llamaremos `introduccion-a-ruby`. No agregaremos espacios para hacerlo más fácil de cargar desde el terminal.

Dentro de la carpeta crearemos una carpeta por unidad. Este módulo tiene 4 unidades, por lo que crearemos `unidad1`, `unidad2`, `unidad3`, `unidad4`.

Comenzaremos trabajando en la `unidad1`.

Podemos dejar la carpeta `introduccion-a-ruby` bajo versionamiento con GIT pero no la subiremos a Github. Reservaremos esta opción para nuestros proyectos grandes, que queramos mostrar a empresas reclutadoras.

Dentro de cada unidad agregaremos un archivo readme donde guardaremos nuestros apuntes.

# Capítulo: Introducción a la programación

---

## Objetivos

- Entender en qué consiste la programación.
- Conocer la definición de algoritmo y su importancia dentro de la programación.
- Conocer los elementos que forman parte de un diagrama de flujo.
- Conocer la relación entre pseudocódigo y diagramas de flujo.
- Conocer la importancia de la lógica independiente del lenguaje.
- Conocer la importancia del desarrollo del pensamiento lógico.

## El computador como una calculadora inteligente

Pensemos el computador como la versión inteligente de una calculadora. Nosotros le ingresamos datos, ya sea por el teclado, desde un archivo, o desde una página web, el computador los procesa de acuerdo a un conjunto de instrucciones que especificamos y finalmente nos entrega resultados de acuerdo a esa especificación.

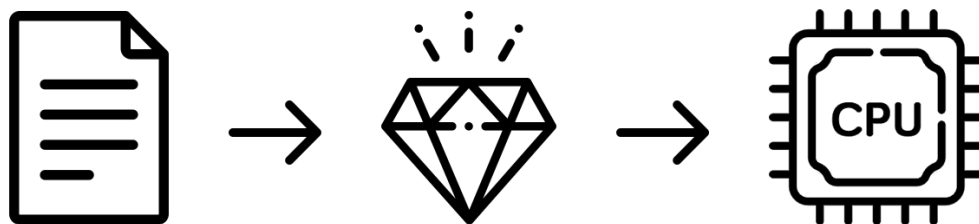
## ¿Qué es programar?

Programar consiste en escribir estas especificaciones paso a paso. Por ejemplo un programa básico podría ser:

Querido computador:

- lee un dato del teclado y guárdalo en el espacio1
- lee un dato del teclado y guárdalo en el espacio2
- suma el dato del espacio1 con el del espacio2 y muestra el resultado al usuario

## Traduciendo las instrucciones



El computador no lee los programas escritos en estos lenguaje directamente. Lo que se hace es ocupar programas compiladores o interpretadores que ponen las instrucciones en términos que pueda entender el computador.

## ¿Qué es un lenguaje de programación?

Un lenguaje de programación es un lenguaje formal que define un conjunto de reglas para escribir instrucciones para un computador.

Existen muchos lenguajes de programación, cada uno con sus ventajas y desventajas para diversos propósitos. Cada lenguaje tiene reglas propias.

## En Ruby

Por ejemplo nuestro programa anterior en Ruby se escribiría:

```
dato1 = gets.to_i  
dato2 = gets.to_i  
print dato1 + dato2
```

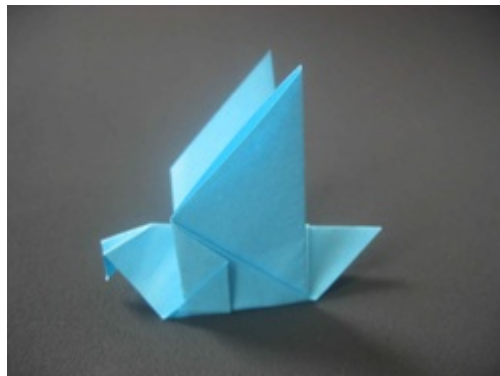
Programar es mucho más que conocer estas reglas.

## El trabajo de un programador

El trabajo de un programador no solo consiste en escribir estos códigos, si no también en entender y poder definir cuál es la serie de instrucciones que nos lleva al resultado deseado. Esto es lo que se conoce como algoritmo.

## ¿Qué es un algoritmo?

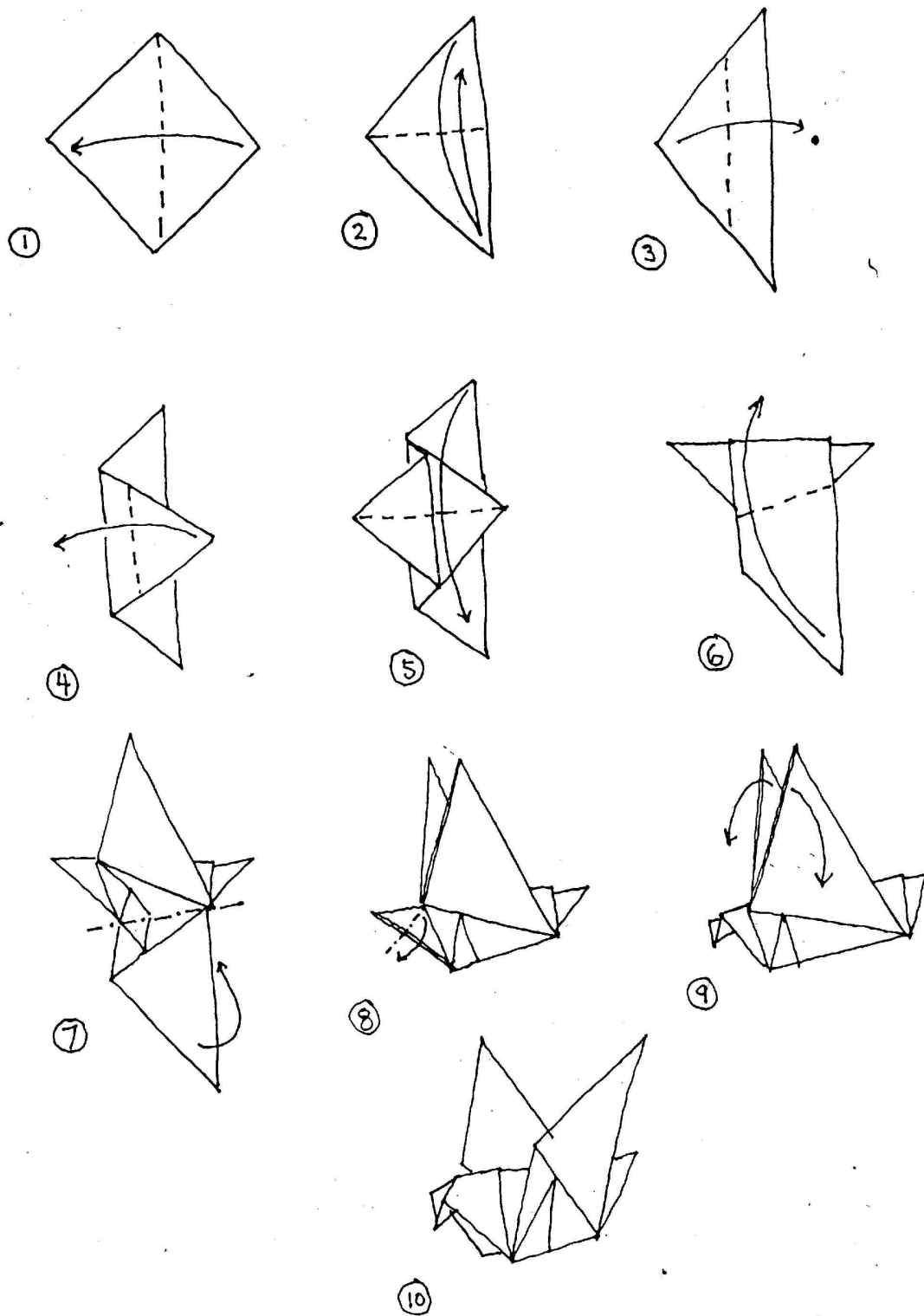
Un algoritmo es una serie finita de pasos para resolver un problema.



Por ejemplo: Para ensamblar un mueble, es necesario seguir todos los pasos del manual de forma secuencial.

Otros ejemplos típicos son una receta de cocina o un modelo de Origami.

## Una serie de pasos para resolver un problema



Flying Bird

Para indicar cómo construir una figura Origami, debemos plasmar claramente cada uno de los pasos; no podemos saltar ningún doblez.

Estos diagramas son de autoría de <http://www.laurenstringer.com>.



## Escribiendo un algoritmo

Escribir un algoritmo es similar a especificar los dobleces. Debemos indicarle a un computador (o a una persona), el paso a paso de cómo resolver el problema. Al igual que en el origami, descubrir cuales dobleces logran los resultados esperados no es trivial y requiere de estudio y práctica.

## Formas de escribir un algoritmo

- Representar en un diagrama de flujo.
- Escribir en pseudocódigo.
- Implementar directamente en un lenguaje de programación.

## Diagrama de flujo

Un diagrama de flujo es una representación gráfica de un algoritmo, ayuda a visualizarlo en casos complejos.



Los diagramas de flujo suelen leerse de arriba hacia abajo y de izquierda a derecha, pero es la dirección de las flechas la que nos indica hacia dónde *fluye* el proceso. Los pasos se leen secuencialmente, o sea el paso 3 sucede después del paso 2 y el paso 2 sólo se ejecuta después del paso 1.

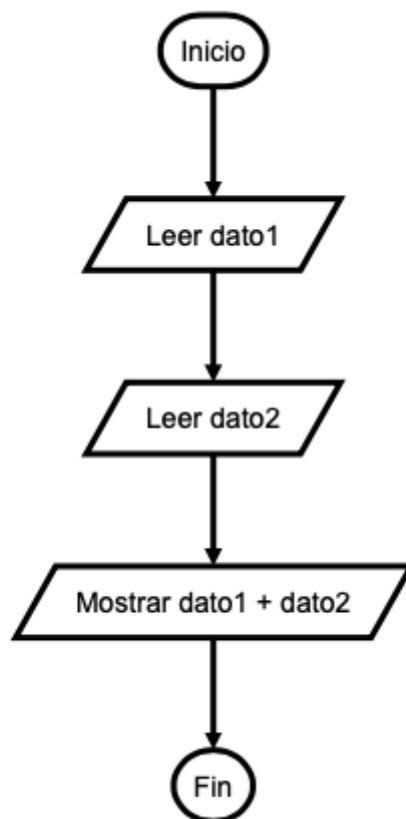
## Símbolos de un diagrama de flujo

Los diagramas de flujo tienen varios símbolos, pero los 4 más utilizados son:

- Inicio y fin del procedimiento.
- Entrada y salida de datos.
- Procesos (instrucciones que ejecuta la máquina).
- Decisiones.



## Escribiendo nuestro programa como diagrama de flujo



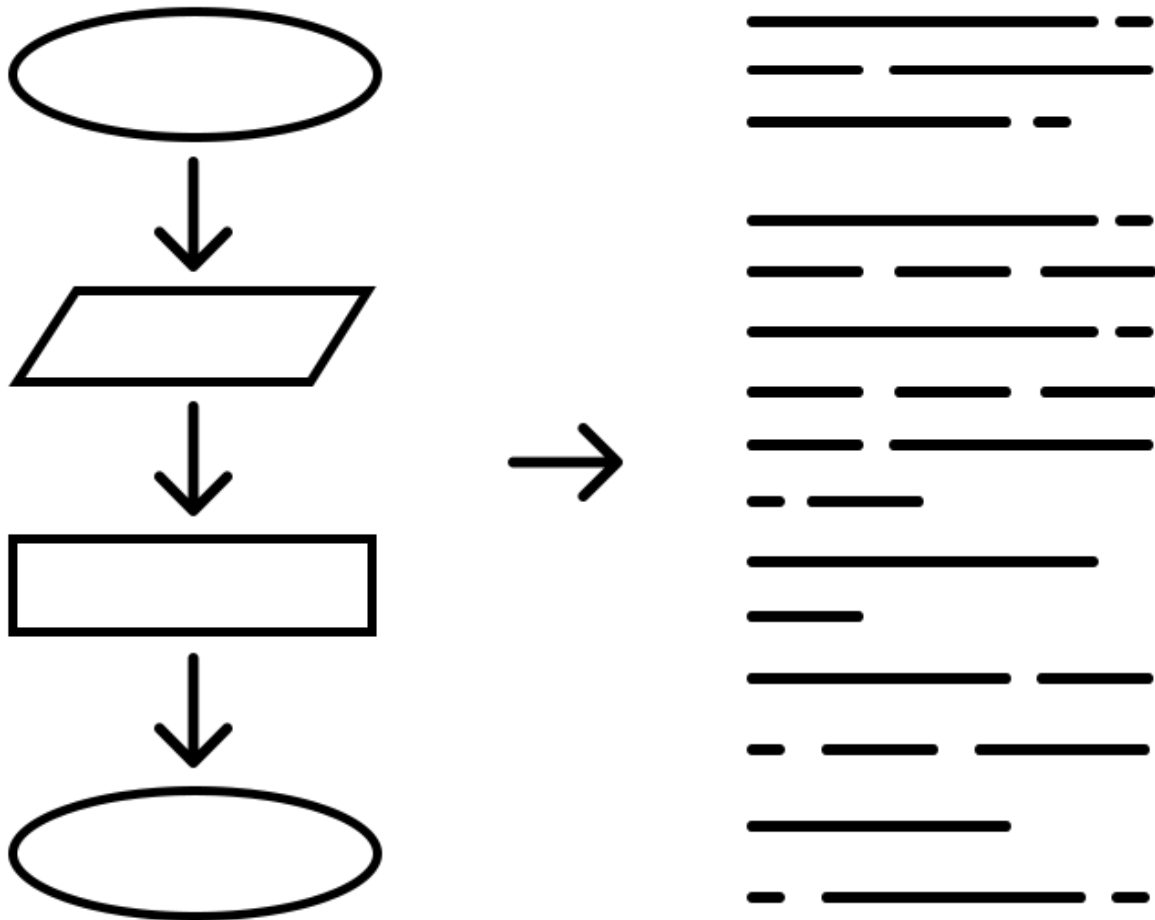
## Pseudocódigo

Es otra forma de representar un algoritmo, diseñada para la lectura.

```
Algoritmo Suma
  Leer dato1
  Leer dato2
  Mostrar dato1 + dato2
FinAlgoritmo
```

En pseudocódigo se utilizan palabras cotidianas, como la instrucción **leer** para especificar que el usuario tiene que ingresar un valor y **mostrar** para imprimir el valor en pantalla.

## Diagramas de flujo y pseudocódigo



Todo diagrama de flujo puede ser transformado a pseudocódigo y viceversa. Todo pseudocódigo o diagrama de flujo puede ser implementado en un lenguaje de programación, aunque el traspaso no siempre será tan directo.

## Ventajas del pseudocódigo

El pseudocódigo, al igual que el diagrama de flujo, nos permite pensar en términos independientes al lenguaje de programación y concentrarnos en describir lo que estamos tratando de hacer y los pasos necesarios en lugar de cómo lograrlo.

En programación, es muy importante poder comunicar al computador lo que tiene que hacer paso a paso. Escribir pseudocódigo y diagramas de flujo ayuda a reforzar esta habilidad.

Otra ventaja es que son independientes del lenguaje, por lo que un algoritmo descrito en pseudocódigo puede ser traspasado a cualquier lenguaje de programación.

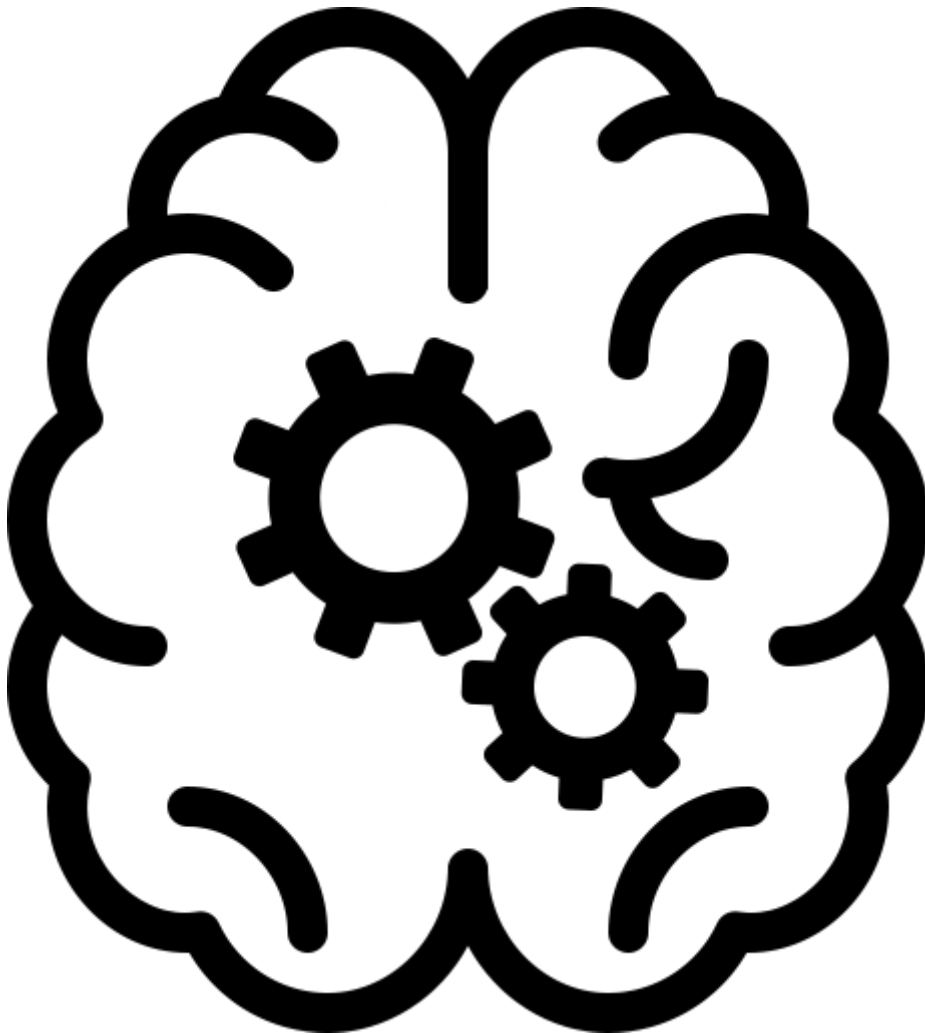
## Enfrentándose a un problema

En programación, al enfrentarnos a un problema, debemos abstraernos para:

1. Analizar el problema.
2. Descomponer el problema en partes que sepamos resolver.
3. Resolver cada parte del problema.

El pensamiento lógico corresponde a una competencia **imprescindible** al momento de programar y está directamente relacionada con nuestra capacidad para solucionar problemas.

## La importancia de desarrollar el pensamiento lógico



Cuanto más desarrollemos nuestro pensamiento lógico, más rápido obtendremos soluciones a nuestros problemas cotidianos en programación.

Para desarrollar estas habilidades es importante que nos demos el tiempo necesario para hacer los ejercicios antes de revisar las soluciones, es normal que al intentar resolver los ejercicios por primera vez demore tiempo. La tolerancia a la frustración y el pensamiento lógico son las competencias claves que te convertirán en un buen programador.

# Capítulo: Introducción a Ruby

---

## Objetivos

- Conocer la historia de Ruby.
- Conocer "posición" de Ruby dentro de los lenguajes más conocidos.
- Conocer ventajas de Ruby sobre otros lenguajes.
- Conocer ecosistema de empresas que utilizan Ruby.
- Conocer distintas versiones de Ruby y cuál utilizaremos en el curso.
- Conocer rol de RVM.
- Instalar RVM en Ubuntu y macOS.
- Instalar Ruby en Windows
- Instalar Ruby utilizando RVM en Ubuntu y macOS.
- Seleccionar una versión de Ruby utilizando RVM.

## Ruby

[Ruby](#) es un lenguaje de programación muy potente y flexible. Fue creado por Yukihiro Matsumoto y su primera versión fue lanzada en el año 1996. Desde entonces, ha sido adoptado por miles de empresas por su capacidad expresiva y cientos de herramientas que son fáciles de incorporar.

## Ruby permite:

- Construir de forma sencilla aplicaciones web con manejo de bases de datos.
- Hacer análisis y visualización de datos.
- Hacer scrapping (captura de datos de una página web).
- Crear juegos de video.
- Construir aplicaciones de escritorio.

Particularmente Ruby es famoso por su capacidad para construir aplicaciones web junto a Ruby on Rails.

## ¿Es Ruby una buena elección para comenzar a programar?

La respuesta es sí.

Ruby es un lenguaje que tiene muchas ventajas interesantes. Está pensado en la felicidad del programador, permite hacer muchas cosas con pocas líneas de código y su sintaxis es tan sencilla que permite ser leído de una forma muy similar al inglés.

## Ruby en acción

Por ejemplo, ¿qué crees que hace la siguiente expresión?

```
3.times { print "hey hey hurray" }
```

O ¿Cuál crees que será el resultado de la siguiente expresión?

```
[1, 2, 3, 4].sum
```

Efectivamente, esta expresión entrega la suma de los números. Esto es lo que convierte a Ruby en una excelente opción para comenzar a programar.

## ¿Es Ruby relevante en la industria de desarrollo?

Ruby es la pieza clave detrás de [Ruby on Rails](#), un framework para construir aplicaciones web. Hoy en día existen muchos lenguajes y frameworks para este propósito, las diferencias suelen ser bastante amplias en lo que respecta a sus propósitos y usos.

## ¿Qué empresas utilizan Ruby o Ruby on Rails?

Estas son algunas de las empresas más conocidas pertenecientes al ecosistema de empresas que utiliza Ruby on Rails como framework en su plataforma:

1. [Airbnb](#)
2. [Github](#)
3. [SlideShare](#)
4. [Soundcloud](#)
5. [Couchsurfing](#)
6. [Groupon](#)
7. [Kickstarter](#)
8. [Twitch](#)
9. [Scribd](#)
10. [Shopify](#)

Incluso la [plataforma académica](#) que ocuparemos para este curso fue desarrollada en Ruby on Rails.

## ¿Por qué aprender Ruby y no otro lenguaje de programación como Java, C# o JavaScript?

Todos los lenguajes de programación tienen sus ventajas, y con el tiempo y la experiencia estas diferencias comienzan a notarse más.

Ruby tiene varias ventajas sobre muchos otros lenguajes:

- Tiene una comunidad muy activa, unida y dispuesta a ayudar.
- El lenguaje está pensando para ser fácil de leer y de escribir.

- Existen miles de componentes fáciles de integrar, sistemas de autenticación, construcciones de gráficos, formularios, manejo de sprites para videojuegos, manejo de base de datos, etc.

Estas ventajas en conjunto permiten a equipos pequeños de desarrollo construir grandes proyectos.

## Desventajas de Ruby

Ruby también tiene una desventaja, y es que no es un lenguaje particularmente eficiente. Por ejemplo si quisiéramos hacer millones de cálculos matemáticos sería mejor trabajar con `C`.

## Ruby y sus versiones

Tenemos que saber que Ruby ha tenido muchas versiones, y esto es importante porque las versiones no son compatibles entre ellas.

## Breve historia de las versiones de ruby

Version	Latest teeny version	Initial release date
1.0	NA	1996-12-25
1.8	1.8.7-p375[52]	2003-08-04
1.9	1.9.3-p551[56]	2007-12-25
2.0	2.0.0-p648[60]	2013-02-24
2.1	2.1.10[62]	2013-12-25
2.2	2.2.10[68]	2014-12-25
2.3	2.3.7[72]	2015-12-25
2.4	2.4.4[75]	2016-12-25
2.5	2.5.1[77]	2017-12-25
2.6		2018-12-25
3.0		2020

## Versionamiento semántico

Ruby a partir de la versión 2.1 ocupa versionamiento semántico. Esto quiere decir que:

- El primer número indica un cambio mayor, los proyectos no serán compatibles entre versiones mayores.

- El segundo número indica un cambio menor, actualizar esto puede requerir un esfuerzo menor, pero si el proyecto es muy grande de todas formas implicará un gran esfuerzo.
- El último número indica un cambio pequeño, usualmente consiste en parches de seguridad que no romperán el funcionamiento de nuestro proyecto.

## ¿Qué versión utilizaremos?

En este módulo utilizaremos la versión **2.5.3** debido a que es la última versión estable disponible. Es importante, al momento de buscar documentación, utilizar versiones superiores a la 2.0 ya que el cambio es muy radical respecto a las anteriores y muy pocas cosas funcionarán.

## No confundir Ruby con Ruby on Rails

Por otro lado no tenemos que confundir Ruby, el lenguaje de programación, con Ruby on Rails, el *framework* para construir aplicaciones web. Cada proyecto tiene sus propias versiones. Rails va en la versión 5.2

## El problema de una única versión

Porque no siempre estaremos trabajando en proyectos desde cero y, a veces, tendremos que trabajar sobre proyectos ya armados para agregar alguna funcionalidad o corregir un bug.

También veremos que muchos proyectos fijan una versión de Ruby para trabajar.

## Ejemplo:

Imaginemos que estamos trabajando en 3 proyectos de manera simultánea:

1. Proyecto de un cliente que se encuentra en Ruby 2.3 donde debemos agregar una funcionalidad nueva.
2. Proyecto que queremos revisar que se encuentra en Ruby 2.4.
3. Proyecto propio que estamos desarrollando en Ruby 2.5.

## ¿Cómo podemos manejar distintas versiones de Ruby?

Un gestor de versiones resuelve el problema. Este es un programa que permite tener instalado varias versiones de Ruby.

## Existen varios gestores

- RVM (Ruby Version Manager)



- Rbenv
- Chruby

Todos estos tienen el mismo propósito, pero su uso es ligeramente distinto. Nosotros trabajaremos con RVM.

## Desventaja del gestor

No hay desventaja de ocupar un gestor, pero sí es un poco más difícil de instalar. Una posible desventaja es que no existe para el sistema operativo Windows.

## Instalando Ruby en Windows

Instalar Ruby en el sistema operativo Windows es bastante sencillo:

1. Ingresar a <https://rubyinstaller.org/>
2. Descargar el instalador
3. Seguir las instrucciones

Ruby funciona bien en Windows pero no todas las gemas de Ruby funcionan.

## Instalando RVM y Ruby en Ubuntu y macOS

La instalación requiere de varios pasos. Las instrucciones actualizadas las podemos encontrar en <http://instalarrails.com/>

## Seleccionando una versión de Ruby en Ubuntu y macOS

Una vez finalizada la instalación podemos listar la(s) versión(es) de Ruby instalada(s) con la siguiente instrucción:

```
rvm list
```

Si la versión se encuentra listada podemos seleccionarla con `rvm use versión`

```
rvm use 2.5.3
```

Si no se encuentra podemos instalarla con:

```
rvm install 2.5.3
```

## Versión por defecto

Se puede dejar una versión de Ruby por defecto especificando `--default`

```
rvm use 2.5.3 --default
```

Al seleccionarla deberíamos ver en la terminal lo siguiente:

Using ~/.rvm/gems/ruby-2.5.3

# Capítulo: Programando con Ruby

---

## Objetivos

- Conocer y utilizar IRB para ejecutar código Ruby.
- Conocer limitaciones de IRB para ejecutar código Ruby.
- Ejecutar scripts de Ruby desde línea de comandos.
- Leer la salida de un script ejecutado.

## Formas de trabajar con Ruby

Existen distintas formas de utilizar Ruby.

1. A través de un programa llamado Ruby Interactivo o [IRB](#)
2. Utilizando nuestro editor de texto y ejecutándolo con Ruby

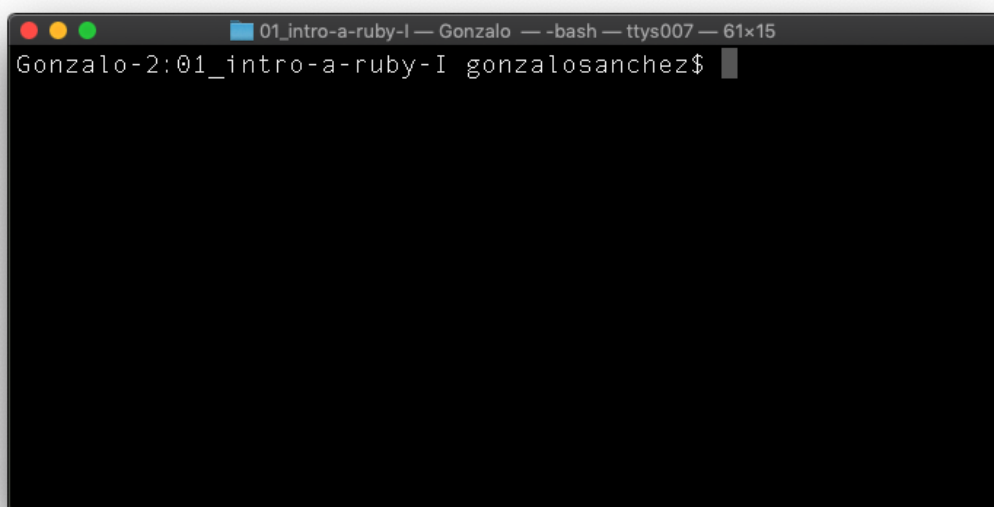
Con IRB podemos escribir código Ruby directamente en nuestro terminal e ir evaluando instrucciones, lo cual es muy útil al momento de realizar pruebas pero es ineficiente para escribir un programa completo.

Con el editor de texto podemos escribir el programa en un archivo .rb y luego ejecutarlo con ruby. **Esta es la forma tradicional de trabajar.**

En esta unidad aprenderemos a trabajar de ambas formas.

## Trabajando con Ruby a través de IRB

Para ingresar a IRB debemos entrar a la terminal.

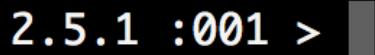


Para eso abriremos la aplicación `Command Prompt` en Windows o `terminal` en Linux y OSX. Al abrirla veremos un signo peso `$` o en algunas ocasiones un signo `>` ó `>>`, símbolo que corresponde al prompt y corresponde al texto que indica que la terminal está lista para recibir una instrucción.

## Ingresando a IRB (Ruby interactivo)

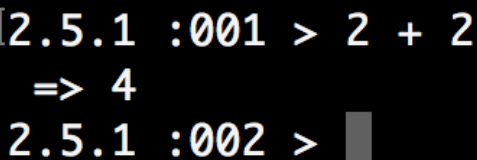
Desde la terminal podemos ingresar a Ruby interactivo. Para ello escribiremos `irb` y luego enter.

Si todo funciona de manera correcta, deberíamos ver que el prompt cambió. Esto es porque ahora estamos ejecutando otro programa: IRB.

A screenshot of a terminal window with a black background. The text '2.5.1 :001 >' is displayed in a light blue or cyan monospaced font. A small grey rectangular cursor is positioned at the end of the prompt.

## Probando IRB

Dentro de este programa podemos escribir instrucciones válidas para Ruby. Por ejemplo, podemos escribir `2 + 2` y ver como resultado `4`.

A screenshot of a terminal window with a black background. The text '2.5.1 :001 > 2 + 2' is displayed in a light blue or cyan monospaced font. Below it, the result '=> 4' is shown in the same color. A second line shows the prompt '2.5.1 :002 >' followed by a grey rectangular cursor.

¿Qué son los otros números que aparecen en la pantalla?

Antes de probar otras operaciones, revisemos algunos detalles que son importantes:

- El número: **2.5.1** corresponde a la versión de Ruby con la que estamos trabajando.
  - Si estás trabajando con la versión **2.5.3** se mostrará ese número.
- El número **:001** indica que es la primera línea que estamos escribiendo. Este número es autoincremental, es decir, irá aumentando a medida que vamos escribiendo y ejecutando instrucciones.
- El resultado de la línea ejecutada aparece acompañado de **=>**

## Realizando nuestras primeras pruebas

Ejecuta, en IRB, las siguientes instrucciones:

- ¿Qué se obtiene al realizar la operación `5 / 2` ?
- ¿Qué se obtiene al realizar la operación `'hola' + 'mundo'` ?
- ¿Qué se obtiene al realizar la operación `[1,2,3].sum` ?

Más adelante -en este módulo- profundizaremos en las instrucciones de Ruby.

## Saliendo de IRB

Para salir de IRB debemos escribir **exit**, y con eso volveremos al prompt del terminal.

## Trabajando con Ruby desde el editor de texto

Hay varios editores de texto gratuitos y muy buenos como [Atom](#), [Visual Studio Code](#) y [Sublime Text](#). Este último es de pago, pero ofrece versión gratuita.

Puedes utilizar el editor que más te guste, y si no sabes cuál escoger, utiliza Visual Studio Code, el cual es el que utilizaremos para los ejemplos de este módulo.

## Extensión de los archivos .rb

Para escribir un programa desde cero en Ruby en un editor de texto, necesitamos crear un archivo con extensión `.rb`

## Nuestro primer programa en el editor

Para escribir nuestro primer programa en un editor de texto vamos a seguir las siguientes instrucciones:

1. Abrir nuestro editor de texto
2. Escribir una instrucción sencilla
3. Guardar el archivo con extensión **.rb**



```
mi_primer_programa.rb x
1 2 + 2
```

## Ejecutando el programa

A continuación, debemos abrir una terminal en la misma ubicación donde se encuentra nuestro archivo y ejecutar `ruby nombre_archivo.rb`

No obtendremos ningún resultado. Es normal.

## Mostrando el resultado

La razón por la que en IRB vimos un resultado, pero no cuando ejecutamos el script desde la terminal es que Ruby no muestra el resultado de cada línea. Si queremos ver este resultado, debemos indicarle a Ruby de manera explícita que muestre el resultado en pantalla, lo cual logramos con la instrucción `puts`.



```
mi_primer_programa.rb x
1 puts 2 + 2
```

## Ejecutando el programa

Si ejecutamos nuevamente el script, veremos el resultado en pantalla:

4

Si utilizamos `puts` directamente en IRB veremos como respuesta `# => nil`. IRB siempre muestra el resultado de la ejecución de la última línea, sin embargo, algunas operaciones (como mostrar en pantalla) no tienen un resultado. `nil` corresponde al objeto nulo, la ausencia de algún valor.

# Capítulo: Elementos básicos de Ruby

---

## Objetivos

- Definir comentarios, variables y constantes en Ruby
- Conocer y utilizar operaciones aritméticas en Ruby
- Manipular strings utilizando concatenación en Ruby
- Manipular strings utilizando interpolación en Ruby
- Manipular entrada y salida de datos utilizando `puts`, `print` y `gets` en Ruby
- Manipular entrada de datos por línea de comandos
- Diferenciar un error sintáctico de uno semántico

## Comentarios

Los comentarios son líneas ignoradas en la ejecución del código, sirven para dar indicaciones y documentar nuestros programas.

```
# Esta línea es un comentario
# Los comentarios son ignorados
# Pueden ir en una línea sin código.
puts 2 + 2 # O puede acompañar una línea de código existente
# puts 4 + 3 Si comentamos al principio todo la línea será ignorada
```

## Comentarios en múltiples líneas

Existe otra forma de hacer comentarios de largo mayor a una línea. Para lograrlo, envolveremos todo el comentario entre un `=begin` y un `=end`

```
=begin
Comentario multilínea:
Ruby
lo
ignoraré
=end
puts "hola"
```

## Importante

A lo largo de este módulo utilizaremos comentarios para mostrar el resultado de una instrucción.

```
puts 2 + 2 # 4
```

Si el resultado de la instrucción no se muestra en pantalla lo mostraremos de la misma forma que IRB, con: =>

```
2 + 2 # => 4
```

## Introducción a variables

- Podemos entender las variables como **contenedores que pueden almacenar valores**.
- Los valores que hay dentro de estos contenedores pueden variar y por eso reciben el nombre de variables.

### Partes de una variable

Una variable se compone de:

- Un nombre
- Un valor

### Asignando un valor a una variable

En Ruby podemos asignar un valor a una variable de la siguiente forma:

```
a = 27
```

En este ejemplo estamos asignando el valor numérico 27 a la variable llamada a

## Introducción a tipos de datos

En Ruby -y en muchos otros lenguajes de programación- existen diversos tipos de dato. Por ejemplo, existen los números enteros, los decimales, fechas, horas, arrays, hashes y muchos otros tipos más.

En este capítulo nos enfocaremos en enteros y strings

### Integers

Los números enteros son números sin decimales que pueden ser positivos o negativos. En inglés se les denomina **Integers**



```
a = 27  
a = -31
```

## Floats

Los números de punto flotante pueden tener decimales.

```
a = 2.5  
b = 3.1
```

## Strings

A una palabra o frase se le conoce como cadena de caracteres o **String**. Un *String* puede estar formado por:

```
a = 'Hola Mundo!' # Una o varias palabras  
b = 'x' # Simplemente un caracter.
```

## ¿Por qué los *Strings* se escriben entre comillas?

Ruby necesita algún mecanismo para diferenciar si el programador está haciendo referencia a una variable o a un *string*.

```
a = 'Esto es un string'  
b = a  
puts b # 'Esto es un string'
```

En la instrucción:

```
b = a
```

Ruby entiende que nos estamos refiriendo a la variable `a` y no a un caracter 'a'

## ¿Comillas simples o dobles?

Para trabajar con *Strings* podemos utilizar comillas simples (') o dobles ("):

```
a = 'Esto es un String'  
b = "Esto también es un String"
```

Se recomienda utilizar comillas simples, con excepción de algunos casos que estudiaremos a lo largo de esta unidad.

## El salto de línea

El salto de línea es un caracter especial que nos permite separar una línea de texto de la siguiente, dentro de un string.

Este caracter es `\n`, indicador de 'nueva línea'

```
a = "hola\na\ntodos"
print a
# hola
# a
# todos
```

`\n` cuando está entre comillas dobles se lee como si fuera un solo caracter.

```
puts "\n".length
puts '\n'.length
```

Esto es porque al usar comillas simples el *string* es entendido tal cual. Al usar comillas dobles se interpretan los *metacaracteres*, o indicadores de símbolos que no se pueden usar normalmente. El `\` en Ruby se ocupa como caracter de escape, esto quiere decir que permite una interpretación alternativa del texto. Existen otras combinaciones interesantes como `"\t"` que permite tabular un texto.

## Operando con variables

Hasta ahora hemos aprendido que una variable puede contener números enteros **Integers**, con decimales **Floats** o palabras **Strings**. Utilizando estos valores podemos realizar operaciones.

### Sumas, restas, multiplicaciones y divisiones

Podemos operar con variables de tipo entero tal como si estuviéramos trabajando en una calculadora.

```
a = 5
b = 2

puts a + 2
# 7

puts b - a
# 3
```

```
puts a * b
#20

puts a / b
# 2
```

De seguro les llama la atención el resultado de esta última operación. Esto es porque Ruby entiende que al dividir enteros el resultado debe ser entero. Por ejemplo, si tenemos dos nidos y cinco pollitos, no podemos dejar 'dos pollitos y medio' en uno de los nidos. Para indicar que queremos el resultado como *float*, debemos expresar alguno de los operandos como tal, de la siguiente forma:

```
a = 5.0
b = 2

puts a / b
# 2.5
```

## Modificando una variable

```
a = 'HOLA!'
a = 100

puts a
# 100
```

Una variable puede cambiar de valor por medio de una nueva asignación.

También podemos modificar el valor de una variable operando sobre ella misma, **este tipo de operación es muy utilizada**:

```
a = 2
a = a + 1 # => 3
puts a # 3
```

## Operando con variables de tipo String

¿Qué obtendremos al "sumar" dos Strings?

```
a = 'HOLA'
b = 'MUNDO'
puts a + b
# "HOLA MUNDO"
```

## Concatenando números

Observemos el siguiente ejemplo:

```
a = '7'  
b = '3'  
  
puts a + b  
#"73"
```

Al asignar los valores utilizando comillas simples, Ruby los interpreta como String. La acción de 'sumar', o sea unir dos o más *Strings* se conoce como **concatenación**, porque enlaza dos *cadena de caracteres*

## Interpolación

Otra acción muy importante y ampliamente utilizada al momento de trabajar con Strings es la **interpolación**.

```
edad = 30  
texto = "tienes #{edad} años"  
puts texto  
# "tienes 30 años"
```

La interpolación es un mecanismo que nos permite introducir una variable (o un dato) dentro un String sin necesidad de concatenarlo. Para interpolar simplemente tenemos que introducir la variable (o dato) utilizando la siguiente notación:

### La interpolación sólo funciona sobre comillas dobles

```
edad = 30  
texto = 'tienes #{edad} años'  
puts texto  
# "tienes #{edad} años"
```

¿Recuerdas que los Strings se pueden declarar entre comillas simple o dobles?

**La interpolación sólo funciona sobre comillas dobles.** Si intentamos interpolar utilizando comillas simples obtendremos el contenido literal.

### La interpolación es más fácil de usar que la concatenación

```
nombre = 'Carlos'
apellido = 'Santana'

# Concatenación
puts "Mi nombre es " + nombre + " " + apellido
# "Mi nombre es Carlos Santana"

# Interpolación
puts "Mi nombre es #{nombre} #{apellido}"
# "Mi nombre es Carlos Santana"
```

Podemos obtener los mismos resultados utilizando concatenación e interpolación, sin embargo, se prefiere la interpolación debido a que es más rápida y presenta una sintaxis más amigable para el desarrollador.

## Constantes

- Existe un tipo especial de variable llamado **constantes**.
- Una constante sirve para almacenar un valor que no cambiará a lo largo de nuestro programa.

En Ruby, la única regla para definir una constante es que su nombre debe comenzar con una letra mayúscula.

### Intentando modificar una constante

```
Foo = 2
# => 2

Foo = 3
# (irb):2: warning: already initialized constant Foo
# (irb):1: warning: previous definition of Foo was here
# => 3

Foo
# => 3
```

### Las constantes pueden modificarse, pero bajo alerta

En este caso vemos que es posible modificar una constante, pero obtendremos un aviso de que estamos haciendo algo mal.

Es convención de que las constantes se escriban completamente con mayúsculas, de esta forma si vemos algo como:

```
NO_ME_CAMBIAS = 1
```

Sabremos que es un error cambiarlo

## Salida de datos

Hay varias formas de mostrar datos en pantalla, las dos más utilizadas son `puts` y `print`. Ambas son muy similares:

- `puts` agrega un salto de línea
- `print` no lo agrega

```
puts "con puts:"
puts "hola"
puts "a"
puts "todos"

#con puts:
#hola
#a
#todos

print "con print:"
print "hola"
print "a"
print "todos"

#con print:holaatodos
```

## Dato interesante

A veces a mostrar le diremos imprimir. Esto tiene razones históricas, de cuando no existían las pantallas y la única forma de mostrar el resultado era imprimirlo. Por eso a veces se dice '*imprimir en pantalla*'.

## Entrada de datos

Con frecuencia nuestro script necesitará interactuar con el usuario, ya sea para seleccionar una opción de un menú o para simplemente ingresar un valor sobre el cual nuestro script va a operar:

Podemos capturar datos introducidos por un usuario utilizando la instrucción `gets`, y de la siguiente forma:

## Ingresando datos con gets

Antes de mostrar el valor, la consola quedará bloqueada hasta que ingresemos una secuencia de caracteres y presionemos la tecla enter.

```
2.5.1 :001 > nombre = gets
Carlos Santana
=> "Carlos Santana\n"
2.5.1 :002 > █
```

El problema del método `gets`, como podemos observar en la imagen, es que también captura el salto de línea que se genera al presionar la tecla enter. En Ruby (y en muchos otros lenguajes) el salto de línea se representa como `\n`

## Removiendo el salto de línea con `chomp`

Para solucionar este problema podemos utilizar `.chomp`:

```
nombre = gets.chomp
puts nombre # "Carlos Santana"
```

## Ingresando datos por consola

A veces, cuando nuestro programa no está pensado para ser usado por humanos, sino por otros programas, es más conveniente que los datos puedan ser entregados como argumentos de línea de comandos. Esto quiere decir que en vez de escribir `ruby programa.rb`, lo invocaremos como `ruby programa.rb dato`.

Estos argumentos se almacenan en un arreglo (una lista, una colección) llamada `ARGV`.

Para obtener estos datos basta con extraer de este arreglo los datos que nos interesan, contando desde el cero en adelante.

Para ponerlo de forma más concreta, hagamos un programa llamado "suma3.rb", que sume tres números ingresados como argumento de línea de comandos:

```
primero = ARGV[0].to_i
segundo = ARGV[1].to_i
tercero = ARGV[2].to_i
puts primero + segundo + tercero
```

Esto nos permitirá llamar al programa y entregarle todos los datos al mismo tiempo, escribiendo `ruby suma3.rb 5 8 12`

Debería mostrarnos el número 25

## Creando un programa con lo aprendido

Con esta simple entrada de datos ya podemos crear programas sencillos. Por ejemplo, un programa que solicite tu nombre y luego te salude:

```
nombre = gets.chomp
puts "Hola #{nombre} !"
```

## Concatenando números por error

Otro detalle interesante es que *gets* siempre entrega un *String*, por lo tanto, si aplicamos operaciones de suma (+) a números ingresados por teclado, estos serán concatenados.

```
num1 = gets.chomp # 4
num2 = gets.chomp # 6
num1 + num2 # 46 :(
```

## Transformando los datos

Este comportamiento se puede modificar aplicando **transformaciones a los tipos de datos**.

```
num1 = gets.chomp.to_i # 4
num2 = gets.chomp.to_i # 6
num1 + num2 # 10 :(
```

La transformación de tipos de datos la estudiaremos en profundidad en un próximo capítulo

## chomp no siempre es necesario

```
num1 = gets.to_i # 4
num2 = gets.to_i # 6
num1 + num2 # 10 :(
```

Si estamos transformando Strings a números, el método `chomp` no es necesario. Prueba el código anterior removiendo el `.chomp`. De todas formas incorporarlo no afecta en el resultado.

## Ingresando un dato al azar

Existe un método para generar un número al azar dentro de un rango.

```
dado = rand(1..6) # Genera un valor entre 1 y 6
```



## Errores sintácticos vs errores semánticos

Los errores podemos clasificarlos en dos tipos: sintácticos y semánticos.

- Los errores sintácticos son muy similares a los errores gramaticales, donde escribimos algo que contraviene las reglas de cómo debe ser escrito.

Por ejemplo: asignar una variable es de izquierda a derecha, `a = 5` mientras lo contrario sería un error sintáctico `5 = a`

- Los errores semánticos son muy distintos, el programa funcionará de forma normal pero el resultado será distinto al esperado. Los errores semánticos corresponden a errores de implementación o de diseño.

## Errores sintácticos

Un error sintáctico será detectado automáticamente cuando Ruby intente leer esa línea, al momento de detectar el error lo reportará y dejará de leer.

## Resumen del capítulo

En este capítulo revisamos varios elementos claves para construir un programa en Ruby, estos son:

- Variables: Contenedores de datos.
- Operadores: Nos permiten sumar, restar y hasta concatenar datos.
- Entrada y salida de datos: Con gets y puts.
- Tipos de dato: Influyen en la forma en que se el computador interpreta la información.
- Interpolación: Forma de insertar una variable dentro de un string.
- Comentarios: Forma de anotar sin que modifique el comportamiento del código.
- Salto de línea: "\n" permite indicar múltiples líneas en un string.
- "" vs ": Las comillas dobles permiten interpolación y caracteres especiales; las comillas simples, no.

# Capítulo: Introducción a objetos

---

## Objetivos

- Leer la documentación de Ruby.
- Conocer la sintaxis de métodos y argumentos.

## Objetos y métodos

Hasta el momento hemos trabajado con tipos de dato **Integer** y **String**, estos tipos de datos reciben el nombre de clases. mientras que los datos de un tipo en específico reciben el nombre de **objetos**

Las operaciones que realizamos sobre ellos reciben el nombre de **métodos**.

## Los métodos nos permiten operar con los objetos bajo ciertas restricciones.

```
2 + 2 # => 4
'hola' + ' a todos' # 'hola a todos'
2 + 'hola' # TypeError: String can't be coerced into Integer
```

Por ejemplo, el método `+` nos permite sumar dos enteros, o concatenar dos strings, pero no nos permite sumar un string con un entero, porque no tiene sentido la idea de 'sumar un texto'.

Para saber cómo ocupar un método en específico o conocer otros métodos debemos ocupar la documentación.

## Revisando la documentación

Consultar la documentación es un hábito que **debemos** adoptar. Todos los programadores la ocupan, incluyendo los expertos.

## ¿Cómo se lee la documentación?

Utilizaremos la documentación de [ruby-doc](#). En esta página podemos ver la documentación con todos los objetos incluidos en Ruby.

Comencemos revisando uno de los objetos que más hemos utilizado: Los [Integers](#).

# Leyendo la documentación

En la documentación veremos 2 columnas:

- La columna derecha nos muestra toda la documentación de los métodos
- La columna izquierda nos muestra 3 secciones:

The screenshot shows the Python documentation for the `Integer` class. At the top is a navigation bar with links: Home, Core 2.5.1, Std-lib 2.5.1, Downloads, a search box, and a Search button. On the left side, there are three sections: 'Home Classes Methods' (with 'Integer' selected), 'In Files' (listing `bignum.c`, `numeric.c`, and `rational.c`), 'Parent' (showing 'Numeric'), and 'Methods' (listing various operators like `::sqrt`, `%%`, `&`, `*`, `**`, `+`, `-`, `~`, `@`, `/`, `<`, `<<`, `<=`, `<=>`, and `==`). The main content area on the right is titled 'Integer' and contains a description: 'Holds Integer values. You cannot add a singleton method to an Integer object, any attempt to do so will raise a `TypeError`.' Below this are sections for 'Constants' (showing `GMP_VERSION` as 'The version of loaded GMP.') and 'Public Class Methods' (showing the `sqrt(n) → integer` method with its description: 'Returns the integer square root of the non-negative integer n, i.e. the largest non-negative integer less than or equal to the square root of n.'). At the bottom of the 'Public Class Methods' section, there is a code block showing examples of the `Integer.sqrt()` method being called with various arguments and returning integer results.

- **In files:** Muestra los archivos donde está definido nuestro objeto.
- **Parent:** Muestra de dónde hereda algunos de los comportamientos, en algunas ocasiones la consultaremos.
- **Methods:** Corresponde a los métodos. Esta será la sección consultaremos con mayor frecuencia.

Al seleccionar cualquiera de los métodos de la tercera sección, seremos redirigidos a la explicación del método en la columna derecha.

## ¿Cómo ocupamos estos métodos?

Existen dos tipos de métodos que se ocupan de forma muy parecida: Los métodos de clase y los métodos de instancia. En la documentación se agrupan por el tipo de método. Por ejemplo, en el caso de la documentación de `integer` vemos que el método `sqrt` aparece bajo `Public class Methods`, mientras los métodos de instancia aparecen bajo `Public Instance Methods`.

## Usando métodos de clase

Para ocupar un método de clase utilizamos la sintaxis nombre del tipo de dato (nombre de la clase) + `.metodo`

Ejemplo:

```
Integer.sqrt(4) # 2  
Time.now
```

## Usando métodos de instancia

Para utilizar un método utilizaremos la sintaxis: `objeto.método` por ejemplo:

```
'paralelepipedo'.size
```

En la gran mayoría de los casos estaremos utilizando métodos de instancia.

## Utilizando el método `.size`

Podemos ver en la documentación que el método `.size` del objeto String devuelve un entero con la longitud del String.

```
'Paralelepipedo'.size # => 14
```

## Definición importante

Utilizar un método es sinónimo de llamarlo o invocarlo, el término más frecuentemente utilizado es **llamar**.

## Métodos con opciones

También existen métodos que requieren recibir información para operar. Por ejemplo el método `.count`, del objeto String, recibe un substring (conjunto de caracteres más corto) para poder contar cuántas veces está contenido ese substring en el String principal.

Imaginemos que necesitamos saber cuántas letras 'p' tiene la palabra 'paralelepipedo':

```
'paralelepipedo'.count('p')  
# => 3
```

## El retorno de un método

Lo que devuelve un método se conoce como el **retorno del método**, y la información que recibe se conoce como **parámetro**.

De tal manera podemos decir que el método `.count`, del objeto String:

- Recibe como **parámetro**, al menos, un substring. Este es obligatorio
- **Retorna** un entero correspondiente a la cantidad de veces que está contenido el substring en el objeto String.

```
'paralelepipedo'.count 'p'  
# => 3
```

## ¿Y los paréntesis?

Cuando utilizamos un método, **el uso de paréntesis es opcional siempre y cuando no haya ambigüedad en cómo se lea.**

```
'paralelepipedo'.count('p')
```

Es lo mismo que:

```
'paralelepipedo'.count 'p'
```

## Algunos parámetros son obligatorios

Dijimos que `.count` exigía un valor, probemos que sucede si no lo ingresamos.

```
'prueba'.count # ArgumentError: wrong number of arguments (given 0, expected 1+)
```

## Existen diversas formas de escribir lo mismo

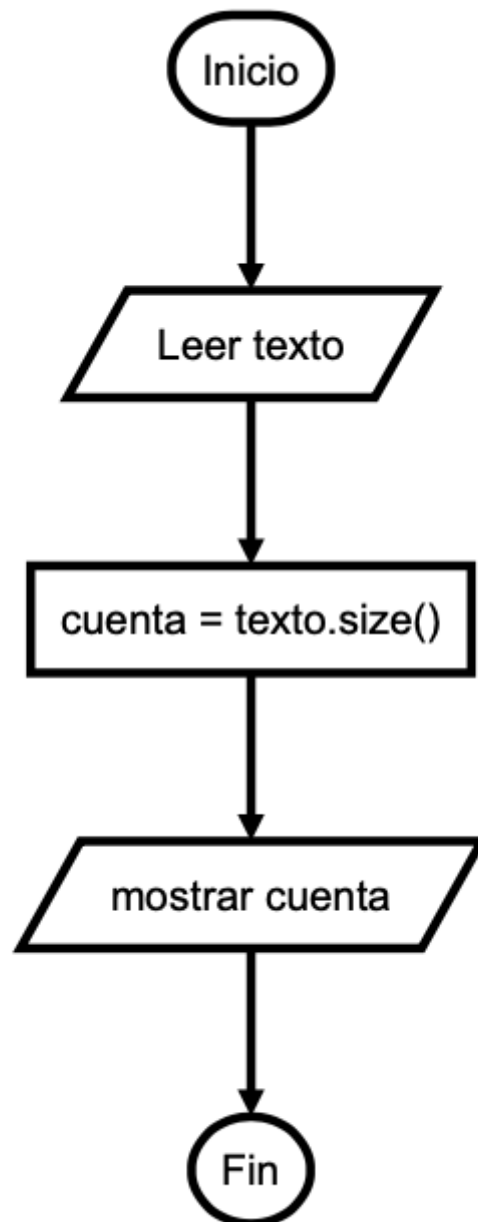
Algunos métodos, como el más (`+`), pueden ser utilizados sin el punto. Por ejemplo `2.+(2)` es lo mismo que `2 +(2)` y, dado que el paréntesis es opcional, esto es lo mismo que `2 + 2`.

## Ejercitando lo aprendido

Lo aprendido lo podemos utilizar para crear un pequeño programa donde el usuario introduce un valor y mostramos la cantidad de letras:

¿Cuál método tenemos que ocupar `.size` o `.count` ?

## Algoritmo



## Código

```
text = gets.chomp # Leer texto
count = text.size # Recordemos que los paréntesis son opcionales
puts "El contenido tiene #{count} letras"
```

Todo ejercicio de programación tiene diversas formas de ser resuelto, por lo que el código de arriba es solo un ejemplo de solución.

## Resumen del capítulo

En este capítulo aprendimos sobre objetos, métodos y cómo consultar la documentación.

- Objetos:
  - En Ruby existen distintos tipos de dato. Un dato de cualquier tipo recibe el nombre genérico de *objeto*. El nombre formal en Ruby para el "tipo de dato" es *clase*.
  - Los objetos tienen métodos que nos permiten realizar diversas acciones sobre ellos.
- Métodos:
  - Corresponden a las formas que tenemos de trabajar con un objeto.
  - Utilizar un método, llamar e invocar son sinónimos.
  - Sólo podemos llamar métodos que ya han sido definidos. De momento sólo hemos trabajado con métodos existentes que consultamos en la documentación, pero más adelante aprenderemos a definir nuestros propios métodos.
  - Los métodos pueden recibir valores (parámetros) y esto los vuelve flexibles.
- Al consultar la documentación:
  - Nombre del método.
  - Tipo de método (de instancia o de clase)
  - Parámetros que recibe: opcionales y obligatorios.
  - Información que retorna.

# Capítulo: Objetos y sus tipos

---

## Objetivos

- Identificar un entero en Ruby
- Identificar un flotante en Ruby
- Identificar un string en Ruby
- Identificar un boolean en Ruby
- Conocer el objeto `nil`.
- Diferenciar el comportamiento del método `+` en Integer vs String
- Transformar Strings a Integers.

## Clases y objetos

En Ruby, existen distintos tipos de dato. Ya sabemos que estos tipos de datos son **clases** y los elementos de un tipo en específico reciben el nombre de **objetos**

## Clases más frecuentes

- Integer: Corresponde a un número entero.
- String: Corresponde a un caracter o una cadena de caracteres.
- Float: Corresponde a un número que puede tener decimales.
- Time: Corresponde a una fecha y hora.
- Boolean: Corresponde a verdadero ( `true` ) o falso ( `false` ). Son el resultado de la evaluación de una proposición lógica.
- Nil: corresponde al objeto nulo, la ausencia de un valor.

A medida que avancemos, profundizaremos más en estas clases.

## ¿Cómo saber de qué clase es un objeto?

Podemos saber el tipo de dato utilizando el método `.class`

Por ejemplo si dentro de **IRB** escribimos `2.class` obtendremos como resultado **Integer**, o si probamos con `'hola'.class` obtendremos como resultado **String**.

```
suma = 5 + 2
# => 7

suma.class
# => Integer

otra_suma = 2.3 + 0.1
otra_suma.class
```



```
# => Float

hora_actual = Time.now
# => 2018-09-10 14:44:24 -0300

hora_actual.class
# => Time
```

## ¿Por qué es importante el tipo de objeto?

Existen distintas reglas para operar entre estos distintos tipos de objetos. Estas reglas las conoceremos consultando la documentación oficial.

Por ejemplo: al sumar dos números obtenemos el resultado de la suma, pero al 'sumar' dos palabras obtenemos la concatenación de estas.

En algunas situaciones, cuando faltemos a estas reglas, las operaciones no serán válidas.

## Concatenando strings

Observemos el siguiente ejemplo: el método `+` del objeto `String` recibe como parámetro otro `String` a concatenar.

 **`str + other_str → new_str`**

Concatenation—Returns a new `String` containing *other\_str* concatenated to *str*.

```
"Hello from " + self.to_s #=> "Hello from main"
```

## Concatenando un string con otro tipo de dato

¿Qué sucede si intentamos concatenar un `Integer` a un `String`?

```
"HOLA" + 2
# TypeError: no implicit conversion of Integer into String
```

¿Y si queremos sumar dos números ingresados por teclado?

```
numero_uno = gets.chomp
# => "10"

numero_dos = gets.chomp
# => "20"

puts numero_uno + numero_dos
# "1020"
```

Para solucionar este problema y, dependiendo de nuestro objetivo, podemos aplicar transformaciones a los tipos de objetos.

## Trasformando tipos de objetos

Existen distintos métodos que nos permiten transformar un objeto de un tipo a otro. Dentro de estos métodos podemos destacar:

- El método `to_i` (To Integer) nos permite convertir un String en un Integer.
- El método `to_s` (To String) nos permite convertir un Integer en un String.

```
2018.to_s
# "2018"

"365".to_i
# 365
```

¡Ahora sí podemos sumar dos números ingresados por teclado!

```
numero_uno = gets.chomp
# => "10"

numero_dos = gets.chomp
# => "20"

puts numero_uno.to_i + numero_dos.to_i
# 30
```

**¿Qué se obtiene como resultado en el siguiente ejemplo?**

```
numero_uno = gets.chomp
# => "10"

numero_dos = gets.chomp
# => "20"

puts (numero_uno + numero_dos).to_i
# ??
```

## Juntando métodos

Veremos de forma frecuente código como el siguiente:

```
numero_uno = gets
numero_uno = numero_uno.chomp
numero_uno = numero_uno.to_i
```

Lo anterior se puede reducir a la siguiente expresión:

```
numero_uno = gets.chomp.to_i
```

La expresión se lee de izquierda a derecha

- **gets** nos devuelve un string con un salto de línea al final
- **chomp** transforma el string en un nuevo string sin el salto de línea
- **to\_i** transforma el string en un número entero
- El resultado es guardado en la variable `numero_uno`

### `.chomp` no siempre es necesario

Recordemos además que el `.to_i` remueve el salto de línea, por lo que nuestra expresión se puede reducir a:

```
numero_uno = gets.to_i
```

## Nota

Más adelante estudiaremos el concepto de *precedencia* y aprenderemos que no toda expresión se lee de izquierda a derecha.

# Interpolación vs transformación vs concatenación

Observemos el siguiente comportamiento:

```
nombre = 'Carlos Santana'
edad = 71

# Concatenación
puts "Hola! Soy " + nombre + " y tengo " + edad + " años!"
# TypeError (no implicit conversion of Integer into String)

# Concatenación + Transformación
puts "Hola! Soy " + nombre + " y tengo " + edad.to_s
# Hola! Soy Carlos Santana y tengo 71 años!

# Interpolación
puts "Hola! Soy #{nombre} y tengo #{edad} años!"
# Hola! Soy Carlos Santana y tengo 71 años!
```

Al utilizar interpolación, no necesitaremos aplicar transformación al objeto `edad`. El método `to_s` será aplicado de forma automática al objeto que escribamos entre las llaves.

# Capítulo: Operaciones aritméticas

---

## Objetivos

- Construir aplicaciones de tipo calculadora, donde el usuario ingresa valores y le entregamos resultados.

Para lograr esto tenemos que:

- Conocer los operadores aritméticos
- Conocer la precedencia de los operadores aritméticos
- Utilizar paréntesis para priorizar operaciones
- Operar sobre números enteros y flotantes

## Introducción a operaciones y operadores

En Ruby existen herramientas que nos permiten, entre otras cosas, sumar, restar, asignar valores, comparar, etc. Todo esto -y mucho más- es posible gracias a los operadores.

## Motivación

¿Por qué debemos aprender de operaciones matemáticas si estamos interesados en crear aplicaciones web o videojuegos?

Los operadores aritméticos los ocuparemos **todo el tiempo**, ya sea para calcular el total de un carro de compras o cambiar la posición de un personaje en un videojuego.

## Operadores aritméticos

Los operadores aritméticos nos permiten realizar operaciones matemáticas sobre números.

Operador	Nombre	Ejemplo => Resultado
+	Suma	$2 + 3 \Rightarrow 5$
-	Resta	$2 - 3 \Rightarrow -1$
*	Multiplicación	$3 * 4 \Rightarrow 12$
/	División	$12 / 4 \Rightarrow 3$
**	Potencia	$2 ** 4 \Rightarrow 16$

## Operaciones con variables

El proceso es exactamente igual si guardamos los valores en variables

```
a = 2
b = 3
puts a + b # 5
```

## Creando un calculadora

Esto nos permite que el usuario ingrese los valores, transformarlos a números y luego operar.

```
a = gets.to_i
b = gets.to_i
puts "a + b es: #{a + b}"
puts "a * b es: #{a * b}"
```

Este código podemos guardarlo en el archivo `calculadora1.rb` y ejecutarlo con `ruby calculadora1.rb`

## Precedencia de operadores

Un concepto muy importante que debemos conocer es el de precedencia. Dicho de otro modo, **saber en qué orden se realiza un grupo de operaciones** .

### Ejemplo de precedencia

Por ejemplo, en la siguiente instrucción ¿cuál es el resultado?

```
10 - 5 * 2
```

$$10 - 5 * 2$$

$$10 - 10$$

$$0$$

10 - 5 \* 2 # 0

## Orden de las operaciones

Veamos una tabla simplificada de precedencia. Esta tabla está ordenada de mayor a menor prioridad, esto quiere decir que la operación de exponenciación precede a (se realiza antes que) la suma.

Operador	Nombre
**	Exponenciación (potencia)
*, %	Multiplicación, división y módulo
+, -	Suma y resta

Cuando dos operaciones tienen el mismo nivel de precedencia se resuelven de izquierda a derecha

## Operaciones y paréntesis

Al igual que en matemáticas, los paréntesis cambian el orden en que preceden las operaciones. Dando prioridad a las operaciones que estén dentro de los paréntesis.

(10 - 5) \* 2

$$(10 - 5) * 2$$



$$5 * 2$$

$$10$$

!!! Los paréntesis importan !!!

## Operaciones con números enteros y decimales

Si dividimos números enteros nos encontraremos con una sorpresa.

```
5 / 3 # => 1
```

Esto es muy común, ocurre en casi todos los lenguajes de programación, pero normalmente esperamos una respuesta diferente. Para obtenerla, debemos ocupar otro tipo de dato, el float.

## Float

En el capítulo anterior mencionamos que existía el tipo de dato asociado a los números **decimales**, llamado float.

```
(3.1).class # float
```

## Enteros y floats

La división entre entero y float, o float y entero da como resultado un float.

```
5.0 / 3.0 # 1.6666666666666667  
5 / 3.0 # 1.6666666666666667
```



La división entre floats también es un float.

Los floats son muy importantes dentro de la programación y tienen propiedades curiosas, una de las más importantes es que solo almacenan una representación aproximada de los números.

```
(10 / 3.0)
=> 3.3333333333333335
```

También podemos transformar a float ocupando el método `to_f`, esto será especialmente útil cuando estemos trabajando con variables que contengan enteros.

```
a = gets.to_i # => 1
b = gets.to_i # => 2
puts a / b.to_f # 0.5
```

En algunos casos, también podemos transformar la variable a float desde un inicio ``ruby a = gets.to\_f

## Ejercicios Resueltos

Resolveremos ejercicios

### Ejercicio de Pitágoras

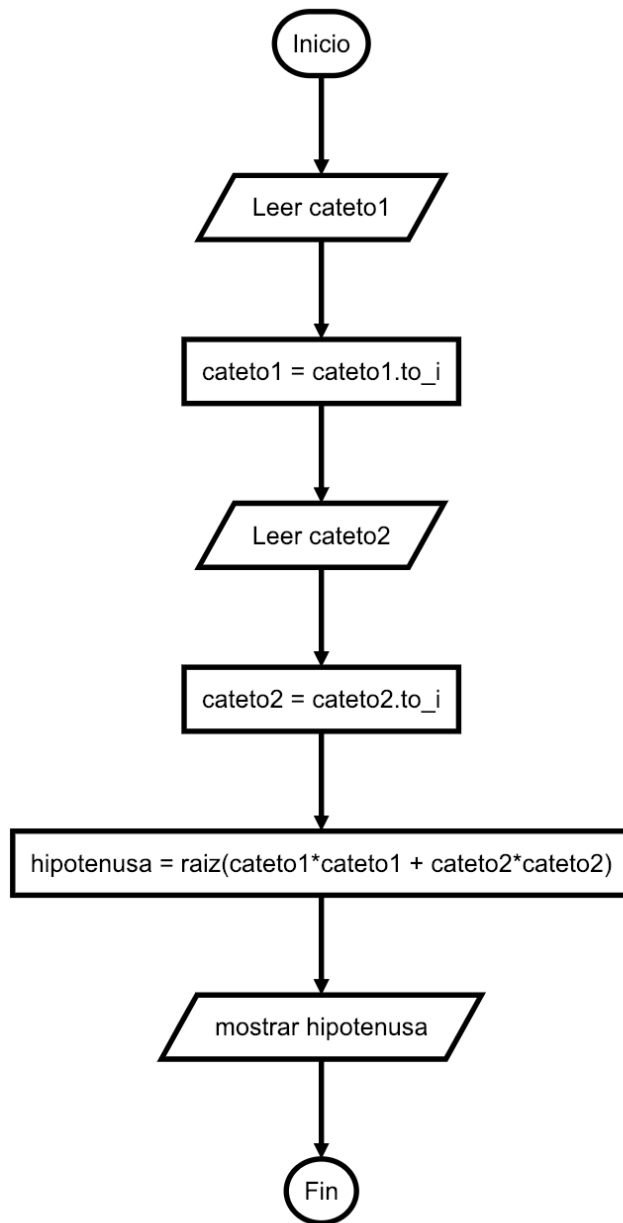
La fórmula de Pitágoras nos permite calcular el largo de la hipotenusa de un triángulo rectángulo a partir de los largos de los catetos. Crearemos un programa donde el usuario introduzca los valores de ambos catetos y entreguemos como resultado el largo de la hipotenusa.

Para implementar el código sólo necesitamos conocer la fórmula.

$$h = \sqrt{c1^2 + c2^2}$$

En Ruby se puede hacer el exponente como `**`, o sea  $a^2$  se puede programar como `a**2`, también es lo mismo que `a*a` las dos opciones son válidas para resolver el problema

## Algoritmo



Al leer un algoritmo tenemos que identificar qué partes no sabemos implementar, o pueden ser complejas. En el caso del algoritmo anterior lo único que no sabemos es calcular la raíz.

Si buscamos en Google `square root y ruby` encontraremos rápido la respuesta.

Para escribir la respuesta crearemos el archivo `pitagoras.rb` dentro de `introduccion-a-ruby/unidad-1` utilizando nuestro editor de código favorito.

## Código

```
c1 = gets.to_i
c2 = gets.to_i
h = Math.sqrt(c1 ** 2 + c2 ** 2)
```

Probemos nuestro programa con `ruby pitagoras.rb` dentro de la carpeta `unidad-1`. Si ingresamos los valores 3 y 4 deberíamos obtener como respuesta 5

Math es un módulo, que en cierto modo es similar a una clase en el sentido de que tiene varios métodos. La documentación de un módulo se lee de igual forma que la de un objeto.

## Fahrenheit

Fahrenheit y Celsius son dos escalas de temperatura frecuentemente utilizadas. Podemos transformar una temperatura de la escala Fahrenheit a Celsius ocupando la siguiente ecuación:

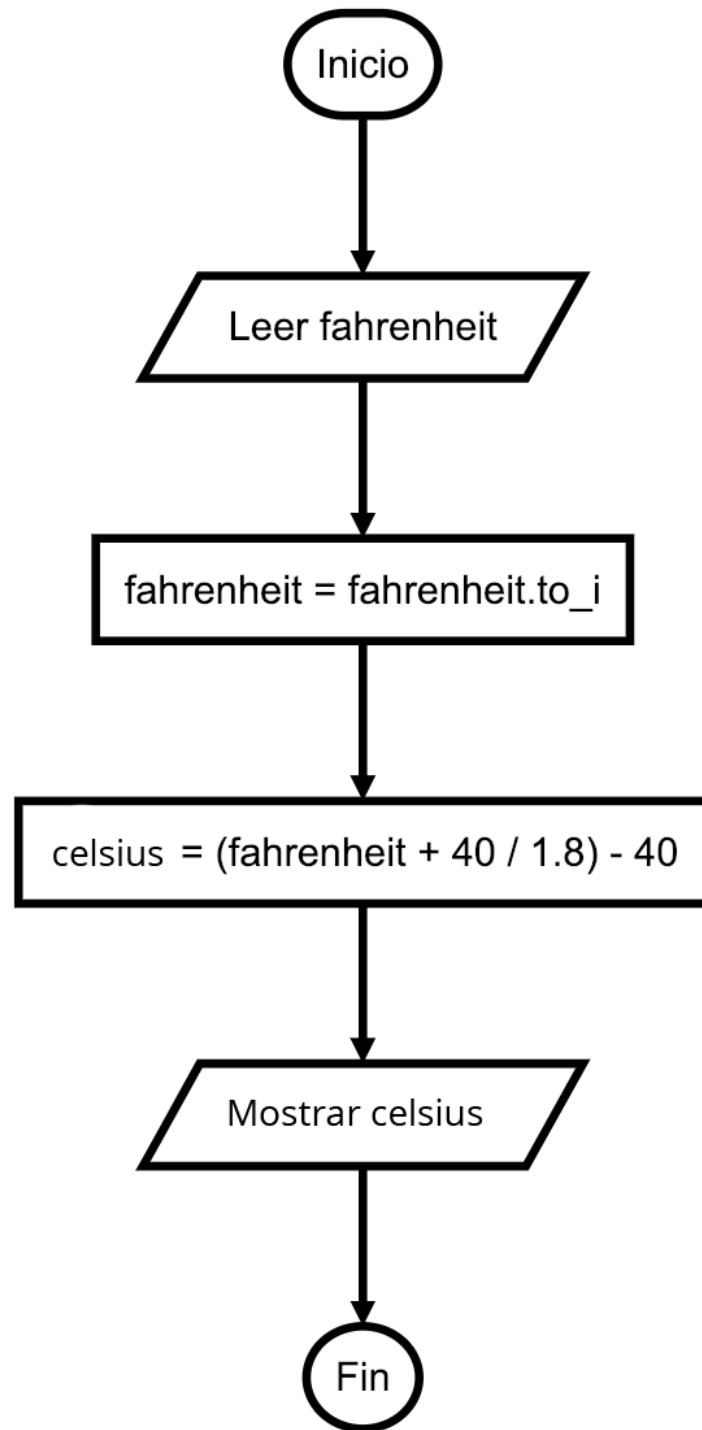
$$\left(\frac{f + 40}{1.8}\right) - 40 = c$$

Antes de desarrollar el algoritmo se deben revisar los siguientes puntos:

- ¿Cuántos valores debe ingresar el usuario? ¿1 o 2?
- ¿Existe algún punto donde tengamos que tener cuidado con la precedencia de operadores?

Luego dibuja el diagrama de flujo y finalmente escribe el código.

## Algoritmo



```
fahrenheit = gets.to_i  
celsius = (fahrenheit + 40) / 1.8 - 40  
puts "la temperatura es de #{celsius} celsius"
```

## Ejercicio 1

Crea el programa `promedio3.rb` donde un usuario debe ingresar 3 valores y se debe calcular el promedio de estos. Recuerda guardar el programa en tu carpeta de estudio.

## Ejercicio 2

Crea un programa `area.rb` donde el usuario ingresa el radio de una circunferencia y se debe calcular el área. Esta se calcula con la fórmula:  $area = \pi * r^2$ .

Recuerda guardar el programa en tu carpeta de estudio.

## Resumen del capítulo

- Precedencia: El orden en que deben ser resuelta las operaciones.
- División entre enteros: El resultado es entero `6 / 4 #=> 0`
- División entre flotante y entero: El resultado es flotante `6 / 4.0 #=> 1.5`
- Paréntesis(): Al igual que en matemáticas nos ayudan a darle prioridad a una operación.

# Capítulo: Introducción a manejo de flujo

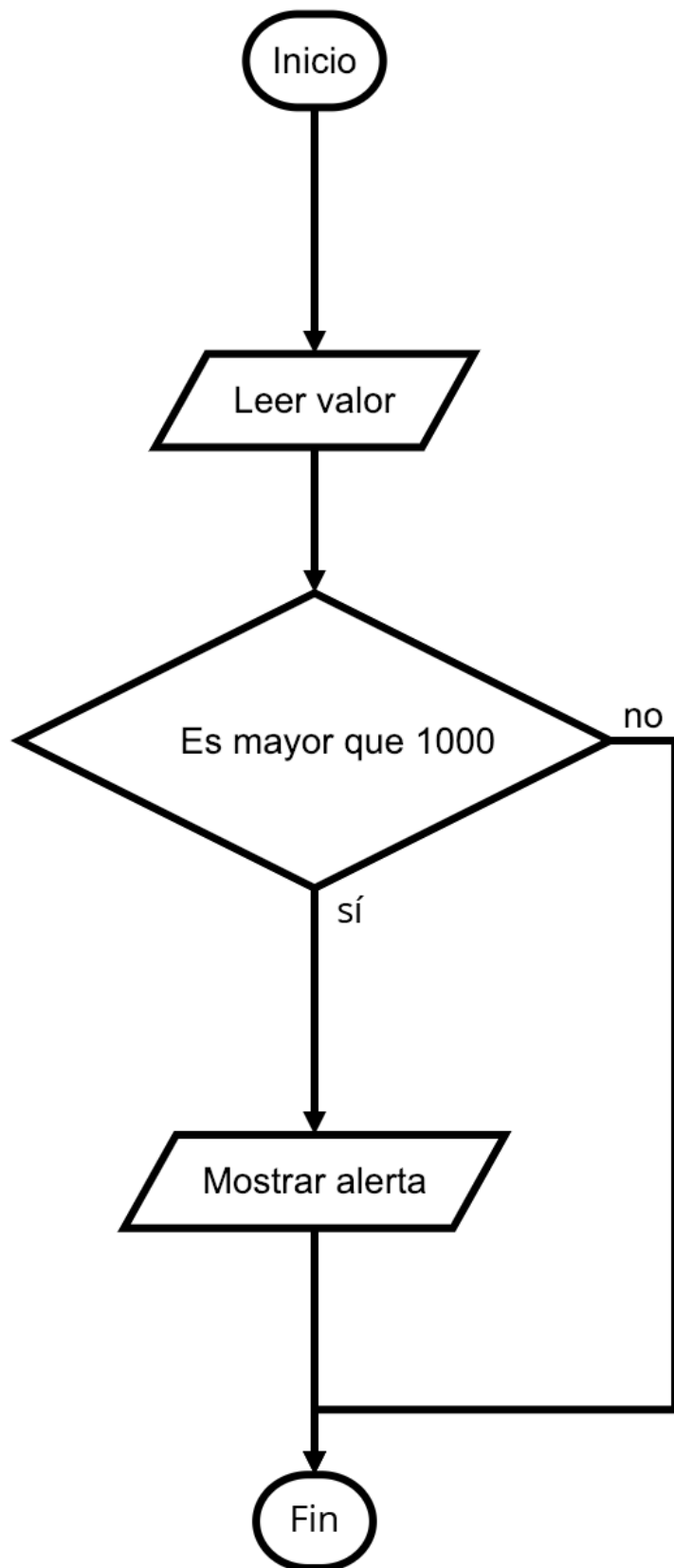
---

## Objetivos

- Crear algoritmos que tomen decisiones en base al valor de una variable.
- Utilizar la instrucción `if` para tomar decisiones.

## Introducción

En algunas situaciones necesitaremos crear algoritmos que tomen una decisión en base a una condición, por ejemplo si un valor es mayor a una cantidad se hace el paso 1 pero si es menor no se hace.



Cuando el algoritmo tiene caminos a seguir entonces empezamos a hablar de flujo.

## La instrucción IF

Ruby -como todo lenguaje de programación- tiene instrucciones para implementar condiciones. la más utilizada es la instrucción `if`.

```
if condition
  # código que se ejecutará SÓLO si se cumple la condición
end
```

Los anterior se lee como: "**Si** se cumple la condición, **entonces** ejecuta el código".

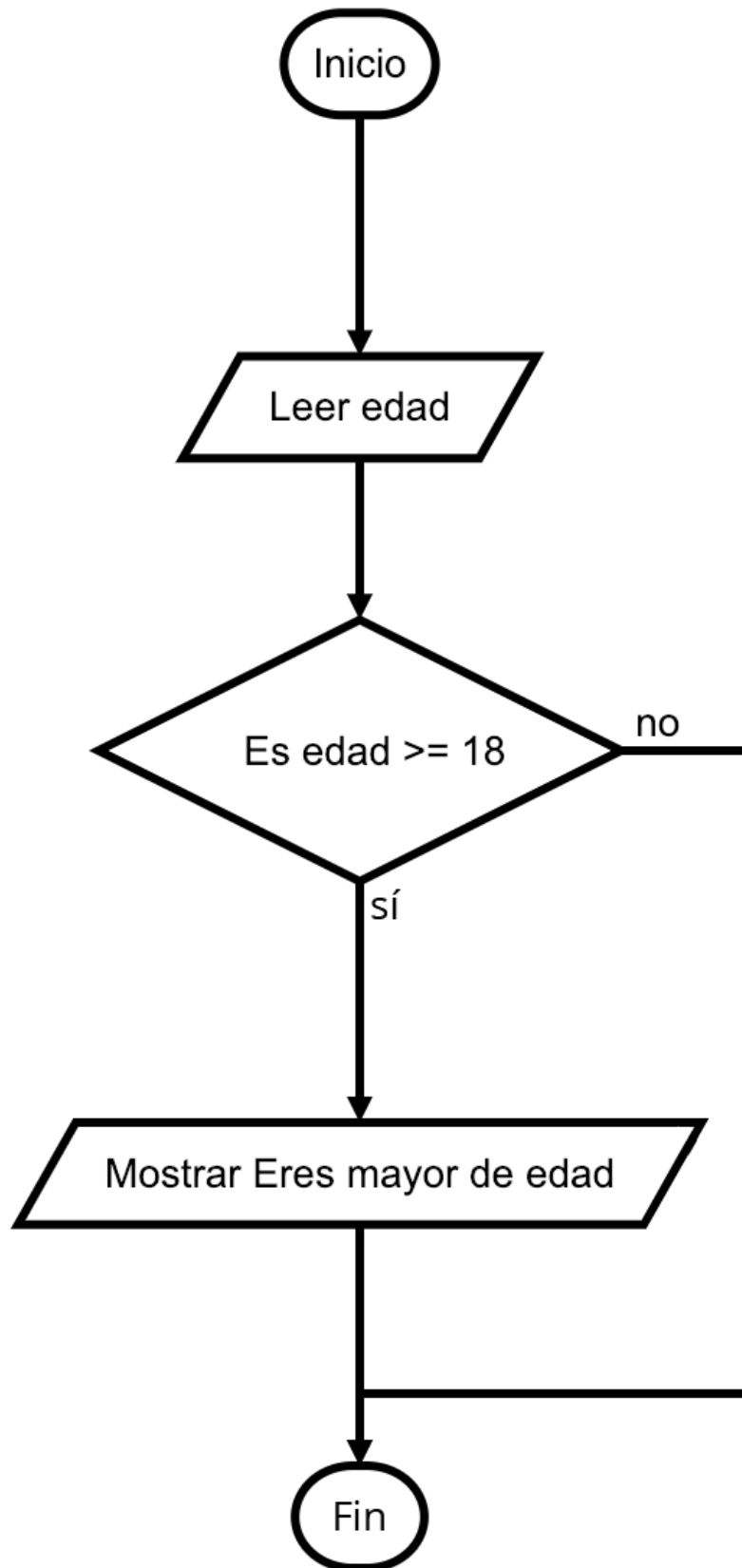
## Ejemplo de if

Analicemos el siguiente ejemplo: En muchos países de Latinoamérica, la mayoría de edad se cumple a los 18 años. Crearemos un programa llamado `mayor_edad.rb` que pregunte la edad al usuario. Si la edad es mayor o igual a 18 entonces le diremos que es mayor de edad.

## Diagrama de flujo

Siempre es bueno ver primero el diagrama de flujo





### Implementemos el código

Dentro de nuestra carpeta de trabajo crearemos el programa `mayor_de_edad.rb` con nuestro editor favorito, donde agregaremos:

```
puts "¿Qué edad tienes?"
edad = gets.chomp.to_i

if edad >= 18
  puts "Eres mayor de edad"
end
```

## Importante

- La instrucción end es muy importante, por cada if tiene que haber un end.
- Todo lo que está dentro del if, sucederá sólo si se cumple la condición.
- El código dentro del if debe estar indentado para poder reconocer de forma sencilla y visualmente dentro del código dónde empieza y termina y que código se ejecuta dentro de la condición.

## Probando el programa

```
Gonzalo-2:examples gonzalosanchez$ ruby ex1.rb
¿Qué edad tienes?
19
Eres mayor de edad
Gonzalo-2:examples gonzalosanchez$ ruby ex1.rb
¿Qué edad tienes?
5
Gonzalo-2:examples gonzalosanchez$
```

Si ejecutamos el programa e introducimos un valor mayor o igual a 18 veremos el mensaje "Eres mayor de edad" en caso contrario, no veremos ningún mensaje.

En este ejercicio comparamos utilizando el operador `>=`, pero existen varios operadores que nos permiten comparar, estos los estudiaremos en el siguiente capítulo.

# Capítulo: Operaciones de comparación

## Objetivos

- Comparar números enteros
- Comparar strings
- Diferenciar comparación de asignación

## Introducción

Las operaciones de comparación son aquellas que nos permiten comparar dos valores y obtener como resultado `true` (verdadero) o `false` (falso).

A este tipo de objeto, resultado de una comparación, se le conoce como **booleano** en honor al trabajo del matemático *George Boole*.

## Operadores de comparación en enteros

Operador	Nombre	Ejemplo => Resultado
<code>==</code>	Igual a	<code>2 == 2 =&gt; true</code>
<code>!=</code>	Distinto a	<code>2 != 2 =&gt; false</code>
<code>&gt;</code>	Mayor a	<code>3 &gt; 4 =&gt; false</code>
<code>&gt;=</code>	Mayor o igual a	<code>3 &gt;= 3 =&gt; true</code>
<code>&lt;</code>	Menor a	<code>4 &lt; 3 =&gt; false</code>
<code>&lt;=</code>	Menor o igual a	<code>3 &lt;= 4 =&gt; true</code>

## Probando las comparaciones en IRB

Realicemos una prueba en IRB, donde el usuario ingrese los valores y veamos si el primero es mayor que el segundo

```
a = 3
b = 4
puts a > b # false
```

## Expresiones más complejas

A veces deseamos operar con los datos que tenemos antes de comparar. Podemos combinar los operadores de comparación con los métodos que deseemos.

```
a = 3
b = 4
puts 2 * a > b + 4 # true
```

En este caso es importante notar que el operador de comparación tiene menor precedencia que cualquier operador aritmético, por lo que primero se evaluarán las expresiones a cada lado del comparador, y luego se compararán los resultados

## Asignación vs Comparación

Un error muy frecuente de principiante es intentar comparar utilizando un signo igual en lugar de utilizar dos.

Con comparación:

```
a = 3
a == 2 # false
```

Con asignación:

```
a = 2
a = 3 # 3
```

## Operadores de comparación en strings

Aunque en la tabla sólo hayamos mostrado números, podemos comparar dos objetos utilizando un operador de comparación:

```
'texto1' == 'texto2' # false
```

## ¿Puede un texto ser mayor que otro?

```
'a' < 'b' # true
```



En este caso la comparación es por orden alfabético, las letras que aparecen primero en el alfabeto son menores que las que aparecen después. Para entenderlo de forma sencilla: Cuando comparamos dos palabras es menor la que aparecería antes en un diccionario.

En el ejemplo vemos que la palabra recycle es menor que la palabra red porque la letra 'c' es menor que 'd'

# Capítulo: *En caso contrario*

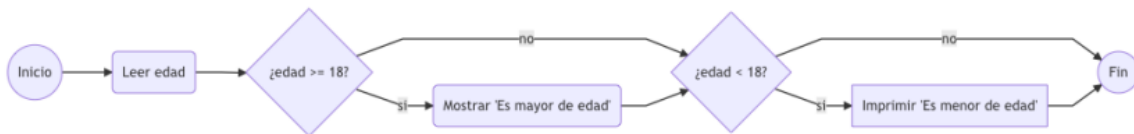
## Objetivos

- Manejar flujos en situaciones de un caso o en caso contrario.
- Utilizar la instrucción `else`

## Introducción

¿Cómo podemos modificar nuestro programa para que muestre un mensaje cuando el usuario sea menor edad y otro mensaje cuando sea mayor de edad?

Una muy **buena práctica** es la de realizar un diagrama de flujo antes de comenzar a programar. Esto ayuda a abstraerse del código y pensar en los pasos críticos.



Transcribamos nuestro diagrama de flujo a código Ruby:

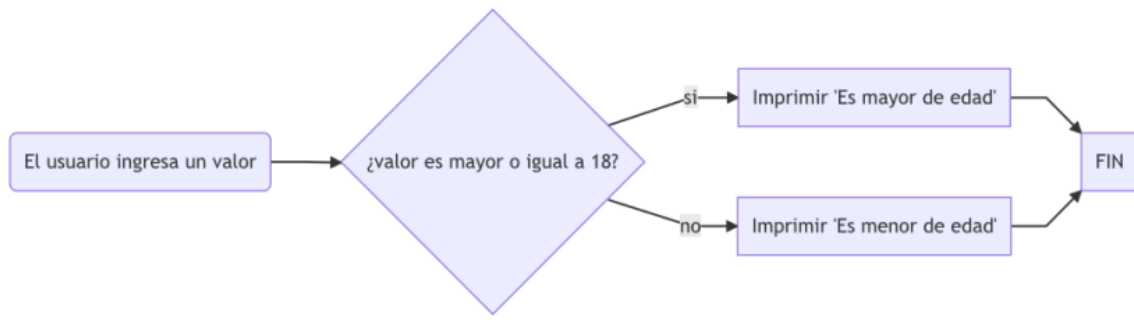
```
puts "¿Qué edad tienes?"
edad = gets.chomp.to_i

if edad >= 18
  puts "Eres mayor de edad"
end

if edad < 18
  puts "Eres menor de edad"
end
```

Una pregunta interesante que nuestro diagrama refleja de manera implícita es: ¿Puede el usuario ser mayor y menor de edad a la vez?

La respuesta es, evidentemente, no. Este tipo de situaciones se puede modelar mejor de la siguiente forma.



Para implementarlo tenemos que introducir una nueva instrucción: `else`

# La instrucción ELSE

La instrucción `else` se utiliza junto con `if` para seguir el flujo de código **en cualquier caso donde no se cumpla la condición**.

```
if condition
  # código que se ejecutará SÓLO SI se cumple la condición
else
  # código que se ejecutará si NO se cumple la condición
end
```

## Implementemos el código de mayor de edad con if y else

```
puts "¿Qué edad tienes?"
edad = gets.to_i

if edad >= 18
  puts "Eres mayor de edad"
else
  puts "Eres menor de edad"
end
```

## Ejercicio resuelto

Crear un programa `mayor_de_2.rb` en nuestra carpeta de trabajo donde el usuario deba ingresar dos números. El programa debe imprimir el mayor de ambos números.

```
puts "Ingresa valor1"
valor1 = gets.to_i
puts "Ingresa valor2"
valor2 = gets.to_i

if valor1 >= valor2
  puts "valor1 #{valor1} es mayor"
else
  puts "valor2 #{valor2} es mayor"
end
```

## Ejercicio

- Crear el programa `password.rb` en la carpeta de trabajo donde el usuario deba ingresar un password en la plataforma, si el password tiene menos de 6 letras se debe mostrar un aviso de alerta que el password es muy corto

## Ejercicio



- Crear el programa `password2.rb` donde el usuario debe ingresar un password, si el password es 12345 se debe informar que el password es correcto pero en caso contrario se debe mostrar que el password es incorrecto.

## Resumen de lo aprendido

- En este capítulo aprendimos a utilizar las instrucciones `if` y `else`, las cuales nos permiten manejar el flujo de un programa
- Existen operadores de comparación que nos permiten comparar si una expresión es verdad o mentira.
- Podemos utilizar los operadores de comparación dentro de la condición de los `if`.
- Debemos tener cuidado de no confundir la asignación de la comparación. Es decir: `=` es distinto de `==`.
- La instrucción `end` es importante: Por cada `if` tiene que haber un `end`.
- Todo lo que está dentro del `if`, sucederá sólo si se cumple la condición.
- El código dentro del `if` debe estar indentado para poder reconocer de forma sencilla y visualmente dentro del código: dónde empieza, termina y qué código se ejecuta dentro de la condición.

# Capítulo: Profundizando en flujo

## Objetivo

- Aprenderemos a resolver problemas donde las posibles alternativas -a una decisión- sean más de dos.
- Analizar situaciones más allá de opción y caso contrario.
- Utilizar `ifs` anidados
- Utilizar la instrucción `elsif` en Ruby

## Abordemos el problema desde un ejemplo

En el capítulo anterior, creamos un algoritmo que determinaba el mayor de dos números ingresados por el usuario. Utilizaremos el mismo ejemplo manejando el flujo cuando ambos números ingresados sean iguales.

Antes:

- caso1: `A >= B` => Decimos que A es mayor
- caso2: `A < B` => Decimos que A es menor

Ahora:

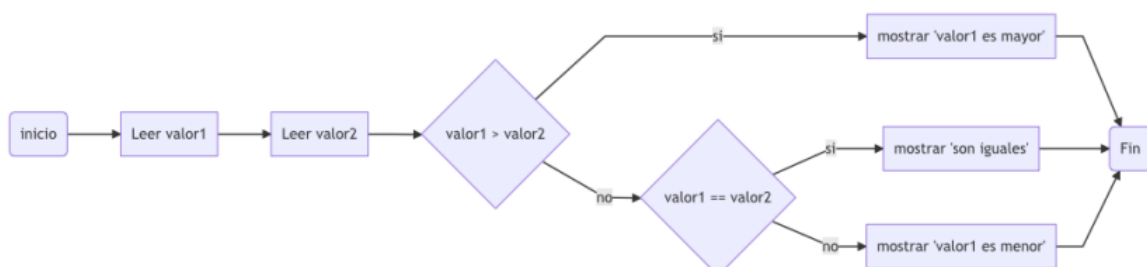
- caso1: `A > B` => Decimos que A es mayor
- caso2: `A == B` => Decimos que A y B son iguales
- caso3: `A < B` => Decimos que A es menor que B

## El mayor de dos números: Escenario donde se ingresan dos números iguales

¿Cómo hacemos para mostrar que ambos son iguales? Ante cualquier problema de este tipo lo mejor es ir paso a paso y analizar el problema.

- Si el primero es mayor -> Mostramos mensaje.
- Si NO es mayor -> Tenemos dos opciones:
  - El primer valor es menor
  - Ambos son iguales

Con esta descomposición del problema, tenemos toda la información necesaria para construir nuestro diagrama:



## Transcribiendo el diagrama

Para escribir este código crearemos el archivo `mayor_menor_o_igual.rb` y dentro agregaremos nuestro código

```
puts 'Ingrese valor1:'
valor1 = gets.to_i #asignación

puts 'Ingrese valor2:'
valor2 = gets.to_i #asignación

if valor1 > valor2 #comparación
  puts "valor1 #{valor1} es mayor"
else
  if valor1 == valor2 #comparación
    puts 'ambos valores son iguales'
  else
    puts "valor2 #{valor2} es mayor"
  end
end
```

Un código siempre debe ser probado en todos los casos, por lo que probaremos el programa 3 veces, ingresando los siguientes valores:

- 2, 1
- 2, 2
- 2, 3

Existe otra solución sin tener que agregar un `if` dentro de otro.

## La instrucción ELSIF

La instrucción `elsif` nos permite capturar el flujo y volver a realizar una evaluación condicional cuando no se cumplió una evaluación previa. Con esta instrucción podemos reescribir el algoritmo del ejemplo anterior, pero conservaremos el archivo y crearemos uno nuevo, llamado `mayor_menor_o_igual2.rb`.

```
puts 'Ingrese valor1:'
valor1 = gets.chomp.to_i

puts 'Ingrese valor2:'
valor2 = gets.chomp.to_i

if valor1 > valor2
  puts "valor1 #{valor1} es mayor"
elsif valor1 == valor2
  puts 'son iguales'
else
  puts "valor1 #{valor2} es menor"
end
```

## Podemos tener tantos elsif como necesitemos

La instrucción `elsif` nos permite continuar realizando tantas evaluaciones condicionales como necesitemos. Finalmente podemos utilizar la instrucción `else` cuando no se cumpla ninguna condición anterior.

Cabe destacar que dos instrucciones `if` **no son lo mismo** que utilizar `elsif`, porque la condición de un `elsif` se evalúa sólo si no se cumple la correspondiente al `if`. `elsif` es en realidad una abreviación de "else if".

## Clasificación según rangos

Es normal estar en situaciones donde nos pidan clasificar algún elemento según un rango.

Por ejemplo supongamos que tenemos una palabra y queremos clasificarla en corta, mediana y larga:

- 4 letras o menos será corta.
- 5 a 10 será mediana.
- Más de 10 será larga.

Crearemos el archivo `clasificador.rb` para crear nuestro código.

```
puts 'Ingresa una palabra'
palabra = gets.chomp
largo = palabra.size

if largo <= 4
  puts 'Pequeña'
elsif largo < 10
  puts 'Mediana'
else
  puts 'Larga'
end
```

## Ejercicio

Modifica el código anterior para poder distinguir palabras muy largas, cuando tengan 15 o más caracteres.

## Solución

```
puts 'Ingresa una palabra'
palabra = gets.chomp
largo = palabra.size

if largo <= 4
  puts 'Pequeña'
elsif largo < 10
  puts 'Mediana'
elsif largo < 15
  puts 'Larga'
else
  puts 'Muy larga'
end
```

## ¿Cuál es la diferencia entre estos códigos?

```
puts 'Ingresa una palabra'
palabra = gets.chomp
largo = palabra.size

if largo <= 4
  puts 'Pequeña'
elsif largo < 10
  puts 'Mediana'
else
  puts 'Larga'
end
```

y:

```
puts 'Ingresa una palabra'
palabra = gets.chomp
largo = palabra.size

if largo <= 4
  puts 'Pequeña'
end
if largo < 10
  puts 'Mediana'
else
  puts 'Larga'
end
```

Probemos con la palabra 'hola', en el primer código obtendremos 'Pequeña', pero en el segundo 'Pequeña' y 'Mediana'

Una vez que una condición `if` o `elsif` se evalúa como verdadera, el flujo entra en esa dirección y no evalúa el resto de las condiciones. Con dos `if` se evalúa el primero y luego el segundo.

# Capítulo: Condiciones de borde

---

## Objetivos

- Conocer la importancia de las condiciones de borde
- Analizar condiciones de borde.
- Evaluar comportamiento de operadores `>`, `>=`, `<` y `<=` en Ruby

## Motivación

En este capítulo aprenderemos a analizar código y buscar errores semánticos.

Es decir, cuando el código está bien escrito pero no hace exactamente lo que queremos que haga. Este tipo de errores es muy típico y es el que más tiempo consume.

La falta de un símbolo, un `;` o algo similar, son errores sintácticos, y son bastante fáciles de detectar. Cuando un programa no funciona como debería, puede llegar a ser bastante más complejo ya que no existe ningún mensaje de error asociado, ni una línea de código en específico que esté fallando.

En este capítulo aprenderemos un tipo de análisis que nos ayuda a descubrir ese tipo de problemas. Se llama *análisis de condiciones de borde*.

Cuando evaluamos los casos específicos alrededor de una condición estamos hablando de **condiciones de borde**. Estos son, precisamente, los puntos donde debemos tener más cuidado.

## Analizando una condición de borde

Revisemos algunos problemas que hemos visto hasta ahora y analicemos sus condiciones de borde.

Por ejemplo, nuestro programa donde se pregunta si una persona es mayor de edad.

```
puts "¿Qué edad tienes?"
edad = gets.chomp.to_i

if edad >= 18
  puts "Eres mayor de edad"
else
  puts "Eres menor de edad"
end
```

En este ejemplo, el punto que define una rama de la otra es el 18. A este punto se le suele llamar **punto crítico**, mientras que los bordes son los números que "bordean" el 18, es decir, analizar los bordes corresponde a probar con los valores **17, 18 y 19**.

En el siguiente algoritmo intenta analizar los bordes:

```

puts 'Ingresa una palabra'
palabra = gets.chomp
largo = palabra.size

if largo <= 4
  puts 'Pequeña'
elsif largo < 10
  puts 'Mediana'
else
  puts 'Larga'
end

```

En este caso, los puntos críticos son 4 y 10, por lo que nuestros puntos a analizar deberían ser 3, 4, 5, 9, 10 y 11

Veamos un caso un poco distinto, en que el punto crítico no es un número exacto:

```

puts 'Ingresa valor1:'
valor1 = gets.chomp.to_i

puts 'Ingresa valor2:'
valor2 = gets.chomp.to_i

if valor1 > valor2
  puts "valor1 #{valor1} es mayor"
elsif valor1 == valor2
  puts 'son iguales'
else
  puts "valor1 #{valor2} es menor"
end

```

En este ejemplo, el caso crítico es cuando ambos valores son iguales, y los bordes son cuando el primer número es mayor que el segundo y cuando el primer número es menor que el segundo.

En este caso podemos escoger un par de números para hacer la prueba, digamos 2 y 2, y luego los bordes serían 3 y 2, y 1 y 2.



# Capítulo: Operadores lógicos

## Objetivos

- Hacer uso de los operadores lógicos para evaluar y simplificar expresiones.
- Invertir una condición.

¿Cómo evaluarías la siguiente expresión?

```
a = 24  
a > 20 y a < 30
```

Para probarla aprenderemos que la expresión `y` se puede escribir como `&&`

```
a = 24  
a > 20 && a < 30 # true
```

¿Cómo evaluarías la siguiente expresión?

```
a = 32  
a > 20 && a < 30
```

```
a = 32  
a > 20 && a < 30 # false
```

Es falsa porque solo cumple uno de los criterios, no ambos.

## Motivación

Los operadores lógicos nos ayudan a simplificar los flujos y a evaluar condiciones más complejas. En este capítulo aprenderemos a utilizarlos.

## Operadores lógicos

Operador	Nombre	Ejemplo	Resultado
<code>&amp;&amp;</code>	y (and)	<code>false &amp;&amp; true</code>	Devuelve true <b>si ambos operandos son true</b> , en este ejemplo se devuelve false.

Operador	Nombre	Ejemplo	Resultado
&#124;&#124;	o (or)	false &#124;&#124; true	Devuelve true <b>si al menos una de los operando es true</b> , en este ejemplo devuelve true.
!	no (not)	!false	Devuelve lo opuesto al resultado de la evaluación, en este ejemplo devuelve true.

Observemos los siguiente ejemplos:

```

nombre = 'Carlos'
apellido = 'Santana'

nombre == 'Carlos' && apellido == 'Santana'
# true

nombre == 'Carlos' && apellido == 'Vives'
# false

nombre == 'Carlos' || apellido == 'Vives'
# true

```

## Identities

Hay varias formas de expresar una afirmación en español, de la misma forma sucede en la lógica y en la programación. Por lo mismo hablamos de identidades.

Veamos ejemplos de esto

### 'Igual' es lo mismo que 'no distinto'

Negar algo dos veces es afirmarlo (en español, no siempre es así; en programación, sí).

Por lo mismo estas dos afirmaciones son equivalentes:

```

a = 18
puts a == 18 # true
puts !(a != 18) # true

```

Son identidades porque para cualquier valor de `a` ambas expresiones siempre se evaluarán igual. Prueba cambiando el valor asignado:

```
a = 17
puts a == 18 # false
puts !(a != 18) # false
```

## Mayor y no menor igual

Un caso similar es la comparación `a > 18`. Decir que `a` **no es mayor a 18**, es decir que es **menor o igual a 18**, (debemos incluir el 18 al negar)

```
a = 18
puts a > 18 # false
puts !(a <= 18) # false
```

```
a = 19
puts a > 18 # true
puts !(a <= 18) # true
```

## Unless

Para ayudarnos a escribir las condiciones siempre en positivo, existe una instrucción que es el antónimo del `if`, esta se llama `unless`: Se lee **a menos que...**

```
unless a <= 18
end

if a > 18
end
```

## Resumen del capítulo

- Operadores lógicos: Son importantes porque nos ayudan a determinar si una expresión es cierta o falsa (la base de la programación).
- Los operadores lógicos nos pueden ayudar a simplificar expresiones.
- Trataremos de escribir siempre las condiciones en positivo.
- no (`a > b`) es lo mismo que (`a <= b`).
- no (`a == b`) es lo mismo que (`a != b`).
- if (`a > 18`) es lo mismo que unless (`a <= 18`).

# Capítulo: Simplificando IFs anidados

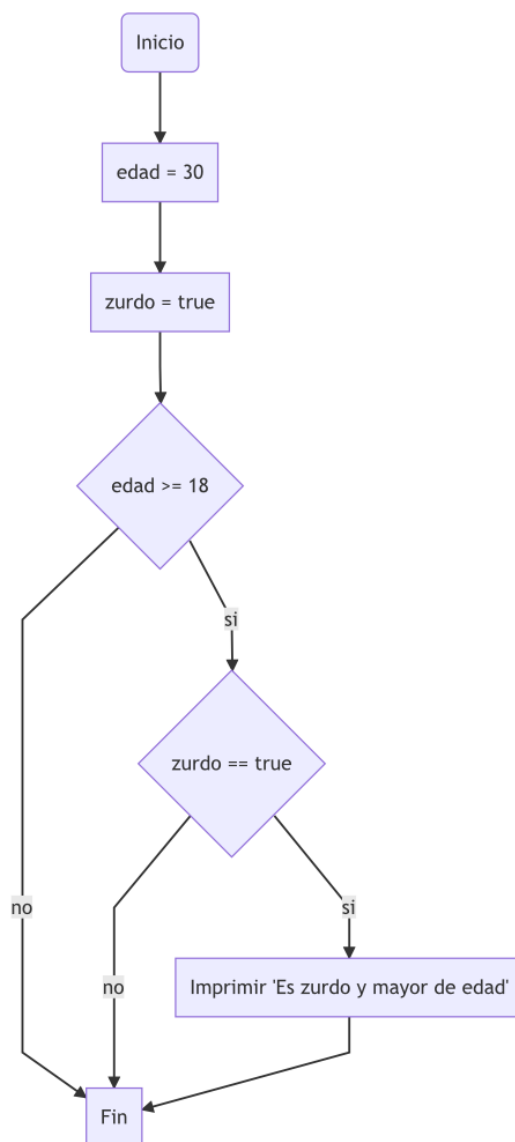
---

## Objetivos

- Simplificar problemas con condiciones anidadas en Ruby
- Entender complicaciones en ifs anidados en Ruby

## Identificando ifs anidados

Analicemos el siguiente ejemplo:



```
edad = 30
zurdo = true

if edad >= 18
  if zurdo == true
    puts 'Es zurdo y mayor de edad'
  end
end
```

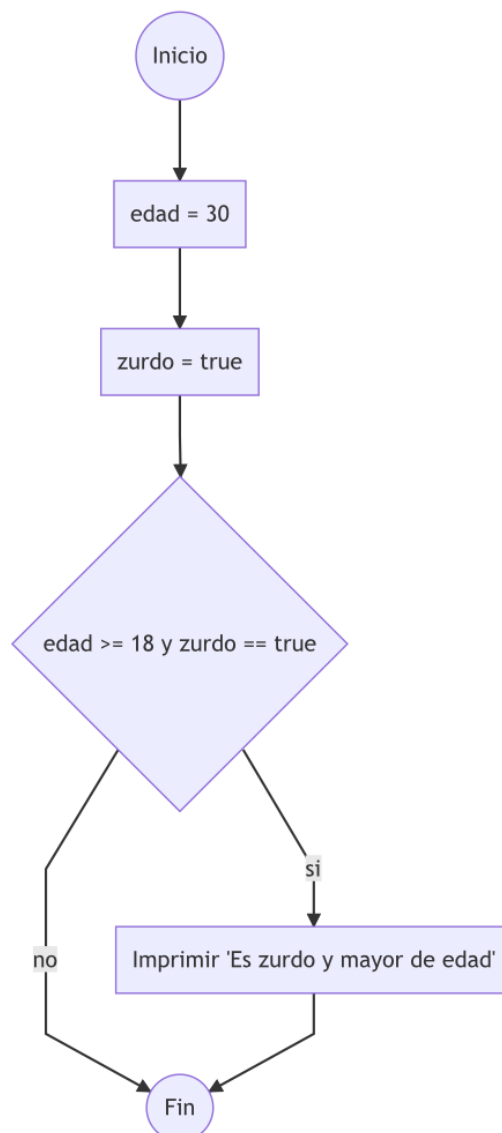
En el ejemplo anterior vemos un `if` dentro de otro `if`: a esto se le llama tener `if` anidados.

El código escrito muestra el texto sólo si se cumplen las dos condiciones.

## Operadores lógicos

Podemos simplificar el código para evaluar ambas condiciones en una misma instrucción `if`. Para una correcta utilización de instrucciones condicionales múltiples, debemos recordar la tabla de operadores lógicos:

¿Qué operador podemos utilizar si necesitamos que **ambas condiciones sean verdaderas**?



```
edad = 30
zurdo = true

if edad >= 18 && zurdo == true
  puts 'Es mayor de edad y zurdo'
end
```

Simplificando condiciones compuestas, el diagrama se simplifica bastante y expresa mejor el flujo del programa. Además, el código resulta mucho más fácil de comprender:

## Ejercicio de integración

Se busca crear un programa que solicite al usuario ingresar tres números. El programa debe determinar el mayor de ellos. Se asume que los números ingresados serán distintos.

¿Cómo lo podemos resolver?

Si bien es posible utilizar ifs anidados, vamos a utilizar lo aprendido para resolverlo de forma inteligente:

El **número 1** es mayor o igual que el **número 2** y mayor o igual que el **número 3**

```
puts 'Ingresa primer número: '
a = gets.to_i

puts 'Ingresa segundo número: '
b = gets.to_i

puts 'Ingresa tercer número: '
c = gets.to_i

if a >= b && a >= c
  puts "a es el mayor"
elsif b >= c
  puts "b es el mayor"
else
  puts "c es el mayor"
end
```

# Capítulo: Simplificando el flujo

---

## Objetivos

- Simplificar el código de un script en Ruby utilizando buenas prácticas y la regla de condición en positivo
- Conocer sintaxis de if ternario en Ruby
- Refactorizar flujo condicional en Ruby

## Variantes de IF

### If en una línea (inline)

En Ruby, también es posible utilizar versiones cortas de la instrucción `if` y `unless` de la siguiente forma: `action if condition`. A este tipo de expresiones se le conoce como *if inline* por estar en la misma línea.

Analicemos los siguientes ejemplos:

```
puts 'Ingresa tu edad: '  
edad = gets.to_i  
  
puts 'Eres mayor de edad!' if edad >= 18
```

```
puts 'Ingresa tu nombre: '  
nombre = gets.chomp  
  
puts "Hola! #{nombre}!" if nombre != ""
```

Esta forma de la instrucción `if` es más limitada ya que no tenemos manejo del flujo cuando no se cumple la condición ( `elsif` y `else` ), sin embargo, es una muy buena solución cuando nos enfrentamos a evaluaciones sencillas como las anteriores.

## Operador ternario

El operador ternario es una variante de `if` que permite operar en base a dos caminos en condiciones simples.

La lógica es la siguiente:

```
si_es_verdadero ? entonces_esto : sino_esto
```

```
edad = 18  
  
puts edad >= 18 ? "Mayor de edad" : "Menor de edad"
```

Lo anterior se lee: Si la edad es mayor o igual a 18, imprime "Mayor de edad"; sino, imprime "Menor de edad".

## Refactorizar

Cuando comenzamos a operar con condicionales es común que caigamos en redundancias innecesarias. Analicemos el siguiente ejemplo:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad == true
  if zurdo == true
    puts "Mayor de edad y zurdo!"
  else
    puts "Mayor de edad pero no zurdo!"
  end
else
  if zurdo == true
    puts "Menor de edad y zurdo!"
  else
    puts "Menor de edad pero no zurdo!"
  end
end
```

Reemplacemos los `if` anidados por condiciones múltiples:

```
if mayor_de_edad == true && zurdo == true
  puts "Mayor de edad y zurdo!"
elsif mayor_de_edad == true && zurdo == false
  puts "Mayor de edad pero no zurdo!"
elsif mayor_de_edad == false && zurdo == true
  puts "Menor de edad y zurdo!"
else
  puts "Menor de edad y no zurdo!"
end
```

Ahora podemos refactorizar las comparaciones en los condicionales que son innecesarias.

## ¿Cómo?

Recordemos que la instrucción `if` espera que el resultado se evalúe como `true` o `false`. Por lo tanto:



```
mayor_de_edad = true

if mayor_de_edad == true
  puts "Mayor de edad"
end
```

Es lo mismo que:

```
mayor_de_edad = true

if mayor_de_edad
  puts "Mayor de edad"
end
```

La comparación `mayor_de_edad == true` es redundante ya que la variable por sí sola se puede evaluar como `true` o `false`. Apliquemos esto en el ejemplo:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad && zurdo
  puts "Mayor de edad y zurdo!"
elsif mayor_de_edad && zurdo == false
  puts "Mayor de edad pero no zurdo!"
elsif mayor_de_edad == false && zurdo
  puts "Menor de edad y zurdo!"
else
  puts "Menor de edad y no zurdo!"
end
```

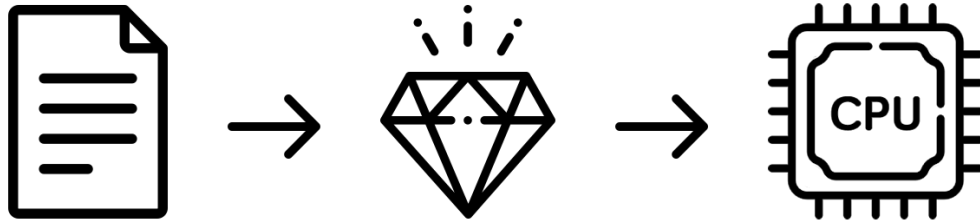
Podemos también refactorizar la comparación para evaluar que una variable booleana sea falsa, simplemente negando la condición:

```
mayor_de_edad = true
zurdo = false

if mayor_de_edad && zurdo
  puts "Mayor de edad y zurdo!"
elsif mayor_de_edad && !zurdo
  puts "Mayor de edad pero no zurdo!"
elsif !mayor_de_edad && zurdo
  puts "Menor de edad y zurdo!"
else
  puts "Menor de edad y no zurdo!"
end
```

## Capítulo: Análisis léxico

En el colegio, en más de alguna ocasión, tuvimos que estudiar la estructura de una oración: Separar el sujeto del predicado, diferenciar los tiempos verbales y todo esto con tal de entender el significado de la oración.



La forma en que un computador lee nuestro código es exactamente igual. Existe un programa encargado de traducir cada una de nuestras instrucciones a un lenguaje que nuestro computador pueda entender.

Conocer los procesos que realiza Ruby al leer un programa nos permitirá entender el por qué son necesarias estas reglas y nos ayudará a memorizarlas.

### ¿Qué sucede cuando ingresamos al terminal y escribimos `ruby mi_programa.rb` ?

Ruby lee y procesa nuestro código.

Detrás de esta lectura y procesamiento se están llevando a cabo una serie de procesos muy interesantes.

No necesitamos conocer estos procesos para poder construir programas en Ruby, sin embargo, conocerlos facilitará nuestra comprensión y nos ayudará a entender de mejor manera los mensajes de error que podemos obtener al ejecutar nuestro código.

## El proceso

El proceso, desde la lectura hasta la ejecución, involucra una serie de etapas que estudiaremos a continuación:

- Tokenización
- Lexing
- Parsing

### Etapas 1: Tokenización

Ruby, al leer un código, lo primero que realiza es una **tokenización**. Esto consiste en separar cada palabra o símbolo en unidades llamadas **tokens**.

Por ejemplo al leer:

```
x = 10
```

Separa la instrucción en 3 tokens:

- x
- =
- 10

## Etapa 2: Lexing

A través de un proceso llamado **lexing** se clasifican los tokens. Cada token es identificado según reglas.

Por ejemplo, como `x` no tiene comillas, Ruby sabe que corresponde a una variable.

## Etapa 3: Parsing

La tercera y última etapa consiste en generar un árbol llamado **árbol de sintaxis abstracta**. Esto corresponde a una forma de representar el código que le permite -a Ruby- saber en qué orden ejecutar las operaciones, o descubrir si todo paréntesis abierto se cierra.

Al terminar la etapa de parsing se genera código **bytecode**. Esto corresponde a código que una máquina virtual Ruby puede leer. La máquina virtual utilizada por defecto se llama **YARV** (Yet Another Ruby Virtual machine) y finalmente es esta la que ejecuta nuestro código.

## Reglas Básicas

Ahora estudiaremos las reglas básicas de Ruby a partir de los tipos de tokens.

Mencionamos anteriormente que en una etapa los tokens son clasificados. Esta clasificación los separa en:

- Identificadores
- Literales
- Comentarios
- Palabras reservadas

## Identificadores

Cuando hablamos de identificadores estamos hablando (para efecto de esta unidad) de **nombres de variables y métodos**.

Para que un nombre -de variable o método- sea válido tiene que seguir ciertas reglas. Un identificador válido es aquel que comienza con una letra de la a a la z o un guión bajo (`_`). Puede ser seguido por letras, guión bajo o números, pero no puede empezar con un número. Los identificadores son sensibles a las mayúsculas. Esto quiere decir que `id` es distinto a `ID`

Por ejemplo, `1a` es un ejemplo de identificador inválido porque comienza con un número. Evaluemos este identificador en **IRB** y veamos qué sucede:

```
1dia = '24 horas'  
# SyntaxError: unexpected tIDENTIFIER, expecting end-of-input
```

El mensaje de error obtenido no puede ser más claro: **tenemos un tIDENTIFIER (o sea un token identificador) no válido.**

Otra regla importante es la sensibilidad a las mayúsculas. Esto quiere decir que un identificador que tenga una mayúscula en su nombre es distinto de uno que no la tiene:

```
gitHub_url = 'https://github.com/'  
# => "https://github.com/"  
  
puts github_url  
# NameError (undefined local variable or method `github_url' for main:Object)  
# Did you mean? gitHub_url
```

Nuevamente el mensaje de error es claro: No existe una variable o método `github_url` definido.

## Literales

Los literales son los valores que asignamos a las variables o simplemente utilizamos para operar:

```
nombre = 'Chuck Berry'
```

`'Chuck Berry'` es un literal. Los literales se llaman así porque representan literalmente el valor.

## Comentarios

Los comentarios, como ya hemos sabemos, corresponden a tokens que serán ignorados por YARV:

```
# Esta línea es un comentario  
# Los comentarios son ignorados  
  
puts 2 + 2 # También pueden existir contiguos a una línea de código válida  
  
# puts 2 + 3 (Esta línea también es ignorada)
```

## Palabras reservadas

Corresponden a ciertas palabras que, como su nombre lo indica, están reservadas para la realización de ciertas tareas. Tienen un significado predefinido y **no se pueden utilizar para nombrar variables**.

Estas son:

BEGIN	class	ensure	nil	self	when
END	def	false	not	super	while
alias	defined	for	or	then	yield
and	do	if	redo	true	
begin	else	in	rescue	undef	
break	elsif	module	retry	unless	
case	end	next	return	until	

Y con esto ya sabemos como un computador puede leer el código de un lenguaje de programación.