

Ruby

Alejandro Chapa Juárez

Programa de Tecnología en Cómputo
Facultad de Ingeniería
Universidad Nacional Autónoma de México

2015

Índice

1	Introducción	3
1.1	¿Qué es Ruby?	3
1.2	Algunas características	3
1.3	¿Por qué usar Ruby?	4
1.4	Primeros pasos	4
2	Hola mundo. Hola Ruby	6
2.1	Tipos básicos	6
2.2	Variables	9
2.2.1	Asignación	11
2.2.2	Constantes	13
2.3	True, false and nil	13
2.4	Operadores de comparación, de igualdad y booleanos	13
2.5	Arreglos	14
2.6	Símbolos	17
2.7	Hashes	18
2.8	Rangos	18
3	Estructuras de control y bucles	19
3.1	Condicionales	19
3.1.1	If, else, elsif	19
3.1.2	Case	22
3.1.3	Unless	24
3.1.4	Operador ternario	24
3.2	Bucles	25
3.2.1	While	25
3.2.2	Until	25
3.2.3	For	26
3.3	Iteradores	26
3.3.1	Iteradores	26
3.3.2	Objetos enumerables	26
3.3.3	Enumeradores	26
4	Funciones	27

<i>ÍNDICE</i>	2
5 Programación Orientada a Objetos	28
6 Programación funcional	29
7 I/O y manejo de Archivos	30
8 Hilos	31
9 Miscelánea Ruby	32
10 Gemas	33

Capítulo 1

Introducción

1.1 ¿Qué es Ruby?

Ruby es un lenguaje de programación de alto nivel interpretado y orientado a objetos que fue creado por el japonés Yukihiro “Matz” Matsumoto en 1993. Matz mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada, y Lisp) para formar un nuevo lenguaje que incorporara tanto la programación funcional como la programación imperativa.

Inicialmente, Matz buscó en otros lenguajes para encontrar la sintaxis ideal. Recordando su búsqueda, dijo, “*quería un lenguaje que fuera más poderoso que Perl, y más orientado a objetos que Python*”.

Matz dice que Ruby está diseñado para la productividad y la diversión del desarrollador, siguiendo los principios de una buena interfaz de usuario y basándose en el principio de *la menor sorpresa*.

Lo primero a mencionar y que es la principal característica de Ruby es que en Ruby todo es un objeto. Absolutamente todo es un objeto, inclusive los números o valores como *true* o *false* son objetos.

1.2 Algunas características

Algunas de las principales características de este lenguaje las vamos a listar a continuación:

- ◇ Las funciones son métodos
- ◇ Las variables siempre son referencias a objetos
- ◇ No requiere declaración de variables
- ◇ Todo tiene un valor
- ◇ Existen iteradores

- ◇ Manejo de expresiones regulares
- ◇ Ruby soporta herencia, pero no multiherencia
- ◇ Ruby tiene variables locales, de clase, de instancia y globales
- ◇ Existe manejo de excepciones
- ◇ Manejo de hilos y multihilos, independiente al sistema operativo
- ◇ Soporta alteración de objetos en tiempo de ejecución
- ◇ La indentación no es significativa, los saltos de línea sí
- ◇ Fácilmente portable
- ◇ Fácil de aprender, poderoso y moderno
- ◇ Multiparadigma
- ◇ Sensible a mayúsculas
- ◇ Cuenta con un recolector de basura

1.3 ¿Por qué usar Ruby?

Ruby es un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla.

Ruby incorpora algunas de las mejores características de lenguajes como Java y Perl, además se promueven las mejores prácticas de programación sin perder la usabilidad. Dinámico e interpretado a la vez.

Simplifica declaraciones, modelos, estructuras sin perder potencia y permite a los programadores que se desarrollen de forma adecuada. Es altamente extensible, a través de librerías escritas en Ruby, las famosas *gemas*.

Su alcance parece ilimitado y en la actualidad se utiliza en diversas aplicaciones, desde desarrollo web (*Ruby on Rails*) hasta la simulación de diversos ambientes.

Por ser un lenguaje multiplataforma se integra perfectamente con diversas arquitecturas, incluso en nuestros dispositivos móviles.

Estas son algunas de las razones por las que amamos Ruby.

1.4 Primeros pasos

Ruby, como ya hicimos mención, es un lenguaje interpretado. Así que necesitamos de un intérprete para poder crear magia con este lenguaje. Para lograr nuestro objetivo haremos uso de IRB (*Interactive Ruby*) el cuál podemos descargar de su página oficial dando clic [aquí](#). Aquí encontraremos las diversas versiones

de Ruby que podemos utilizar y vienen las diferentes plataformas en las cuales podremos trabajar.

La documentación oficial del lenguaje la puedes encontrar en el siguiente link [Documentación Ruby](#) o en esta otra página [Ruby-Doc](#).

Como recomendación podemos usar adicionalmente un editor de texto plano para poder crear nuestros *scripts* en Ruby. Nuestros archivos deberán tener la extensión *.rb* la cual hace referencia a que su contenido contiene sentencias en Ruby.



Figura 1.1: Bienvenido a Ruby

Capítulo 2

Hola mundo. Hola Ruby

Vamos a comenzar con nuestro primer script. Haremos el famoso y obligado “*Hola mundo*”.

Abramos un archivo nuevo y pongamosle de nombre *hola.rb*, el cual tendrá las siguiente líneas como contenido:

```
print "Hola mundo..."
puts "Hola Ruby..."
```

Ahora en nuestra terminal vamos a escribir *ruby hola.rb* con lo cual tendremos una salida como la siguiente:

```
Hola mundo...Hola Ruby...
```

¡Felicidades! Usted ha creado su primer script en Ruby. Como te habrás dado cuenta usamos dos sentencias que parecen realizar lo mismo, *print* y *puts*, pero esto no es así. La principal diferencia que hay entre las dos sentencias que usamos es que *puts* da un salto de línea al final. Ahora cambia el orden, usa primero *puts* y después *print*. Verifica que el resultado es ahora distinto.

2.1 Tipos básicos

Como ya habíamos mencionado, en Ruby todo es un objeto. Así que como tal no existen tipos básicos, si no más bien clases. Aún así vamos a checar un poco sobre los objetos más simples, tales como cadenas y números.

Empecemos con los números...

```
#Esto es un comentario
```

```
#Los números enteros son como los conocemos
```

```
1
```

```
192323
```

```
-500
```

```
#Incluso podemos usar un separador para realizar su lectura más fácil
```

```
1_000 #Es igual a 1000
```

```
#También podemos manejar números en binario, octal y hexadecimal
```

```
0b1111_1111 #255 en binario
```

```
0377 #255 en octal
```

```
0xFF #255 en hexadecimal
```

Veamos un poco de aritmética en Ruby.

```
puts 5+2 #Nos dará como resultado 7
```

```
puts 5-2 #Nos dará como resultado 3
```

```
puts 5*2 #Nos dará como resultado 10
```

```
puts 5/2 #Nos dará como resultado 2
```

Las divisiones entre números enteros, nos darán por resultado un número entero. Si queremos que el resultado tenga decimales debemos usar números flotantes.

```
puts 5.0/2
```

```
puts 5/2.0
```

```
puts 5.0/2.0
```

```
#Las tres operaciones nos devuelve 2.5
```

```
#Un poco más sobre números flotantes
```

```
0.5
```

```
-1.234
```

```
3.23e13 #Esto es 3.23x10^13
```

```
.2 #Atención! Esto no se puede hacer en Ruby
```

```
#debemos poner explícitamente 0.2
```

También contamos con el operador módulo.

```
puts 5 % 2 #Regresará 1
```

```
puts 5 % -2 #Regresará -1
```

```
puts 1.6 % 0.3 #Regresará 0.1, también se pueden usar números flotantes
```

También tenemos exponenciación en Ruby. Veamos algunos ejemplos:

```
x**2 #Es x*x
```

```
x**-1 #Es 1/x
```

```
x**(1/2) #Esto no es raíz cuadrada recordemos que
```

```
#la división de 2 números enteros regresa otro entero
```

```
x**(1.0/2) #Esto sí es una raíz cuadrada
```

En Ruby existen problemas con los números racionales y flotantes, debido al redondeo que se utiliza. Así que lo más recomendable es usar números racionales. Verifiquemos esto con un pequeño ejemplo:

```
puts 1.5-1.1==0.4 #Nos regresará false
```

```
puts 4.0/10==0.4 #Nos devolverá true
```


Los números enteros en Ruby pertenecen a la clase Fixnum, cuando son números pequeños. Y Bignum cuando son números grandes. Los flotantes pertenecen a la clase Float. Esto lo podremos comprobar fácilmente, al ser los números objetos en Ruby, tienen un método con el cual podremos ver a que clase pertenecen.

```
puts 1.class
puts 1.5.class
```

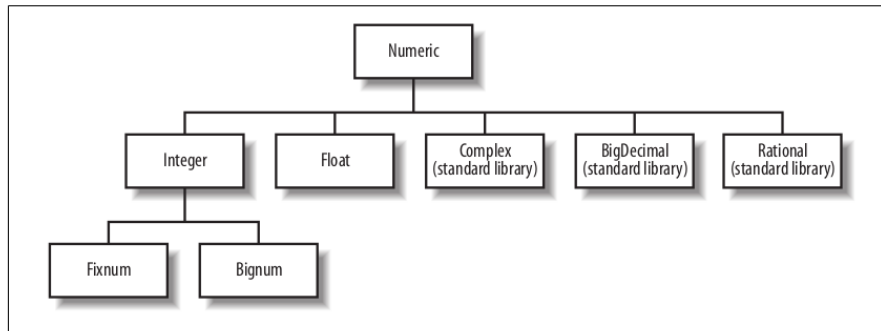


Figura 2.1: Clases numéricas en Ruby

Por último, los números son objetos inmutables, no existen métodos que puedan cambiar su valor. Dejemos en paz un momento a los números, ahora veamos un poco sobre cadenas.

En Ruby podemos escribir cadenas con doble comilla “ o con comilla simple ’. Veamos:

```
puts "Esto_es_una_cadena"
puts 'Esto_también_es_una_cadena'
puts "cadena".class
puts 'cadena'.class
```

Como pudimos observar, ambas cadenas pertenecen a la clase String. Ahora veamos lo siguiente, si una cadena la escribimos usando este símbolo ‘, nuestro intérprete lo reconoce como un comando del sistema, y por ende lo ejecutará y nos arrojará un resultado.

```
puts `ls` #Ejecutará el comando ls en Linux
```

En Ruby existe algo llamado interpolación, el cual es el proceso de insertar el resultado de una expresión dentro de una cadena. La interpolación solo se puede realizar utilizando comillas dobles y se realiza de la siguiente forma:

```
puts 3*2=#{3*2} #Nos imprimirá 3*2=6
```

Las cadenas se pueden concatenar y en Ruby se hace de una manera muy sencilla, solo debemos usar el operador +.

```
puts "Hola"+" "+"'mundo' #Nos devolvera Hola mundo
```

Vamos a dejar las cadenas hasta aquí, mas adelante las abordaremos con mayor profundidad.

2.2 Variables

Las variables las usamos para poder guardar valores que queremos usar posteriormente para hacer cálculos u otras cosas. Una variable puede tener casi cualquier nombre, siempre y cuando no sea alguna palabra reservada. A continuación te dejo una lista de las palabras reservadas que existen en Ruby y cuál es la función de cada una.

Palabra	Función
<i>alias</i>	Crea un alias para un operador, método o variable global que ya exista.
<i>and</i>	Operador lógico, igual a && pero con menor precedencia.
<i>break</i>	Finaliza un while o un until loop, o un método dentro de un bloque.
<i>case</i>	Compara una expresión con una clausula when correspondiente.
<i>class</i>	Define una clase; se cierra con end.
<i>def</i>	Inicia la definición de un método; se cierra con end.
<i>defined?</i>	Determina si un método, una variable o un bloque existe.
<i>do</i>	Comienza un bloque; se cierra con end.
<i>else</i>	Ejecuta el código que continua si la condición previa no es true. Funciona con if, elsif, unless o case.
<i>elsif</i>	Ejecuta el código que continua si la condicional previa no es true. Funciona con if o elsif.
<i>end</i>	Finaliza un bloque de código.
<i>ensure</i>	Ejecuta la terminación de un bloque. Se usa detrás del ultimo rescue.
<i>false</i>	Lógico o Booleano false.
<i>true</i>	Lógico o Booleano true.
<i>for</i>	Comienza un loop for. Se usa con in.
<i>if</i>	Ejecuta un bloque de código si la declaración condicional es true. Se cierra con end.
<i>in</i>	Usado con el loop for.
<i>module</i>	Define un modulo. Se cierra con end.
<i>next</i>	Salta al punto inmediatamente después de la evaluación del loop condicional.
<i>nil</i>	Vacio, no inicializado, invalido. No es igual a cero.
<i>not</i>	Operador lógico, igual como !.
<i>or</i>	Operador lógico, igual a // pero con menor precedencia.
<i>redo</i>	Salta después de un loop condicional.
<i>rescue</i>	Evalua una expresión después de una excepción es alzada. Usada después de ensure.

Tabla 2.1: Palabras reservadas en Ruby

<i>retry</i>	Cuando es llamada fuera de rescue, repite una llamada a método. Dentro de rescue salta a un bloque superior.
<i>return</i>	Regresa un valor de un método o un bloque.
<i>self</i>	Objeto contemporáneo. Alude al objeto mismo.
<i>super</i>	Llamada a método del mismo nombre en la superclase.
<i>then</i>	Separador usado con if, unless, when, case, y rescue.
<i>undef</i>	Crea un método indefinido en la clase contemporánea.
<i>unless</i>	Ejecuta un bloque de código si la declaración condicional es false.
<i>until</i>	Ejecuta un bloque de código mientras la declaración condicional es false.
<i>when</i>	Inicia una clausula debajo de under.
<i>while</i>	Ejecuta un bloque de código mientras la declaración condicional es true.
<i>yield</i>	Ejecuta un bloque pasado a un método.
<i>_FILE_</i>	Nombre del archivo de origen contemporáneo.
<i>_LINE_</i>	Numero de la linea contemporánea en el archivo de origen contemporáneo.

Tabla 2.2: Más palabras reservadas

Bastantes palabras, ya iremos revisando la gran mayoría de ellas. Ahora regresando a las variables, existen diversos tipos dependiendo de su alcance y su entorno, ahorita nos enfocaremos en las *variables locales*. Las variables locales en Ruby deben empezar con letra minúscula o un guión bajo y deben estar formada por letras, números o guiones bajos.

En Ruby existe algo llamado interpretación dinámica lo cual significa que cuando declaramos nuestras variables no debemos de especificar el tipo del cual será.

2.2.1 Asignación

A las variables se les necesita asignar un valor. Para asignar un valor a una variable usamos el operador igual (=).

```
a=1
b=2
c=a+b
```

También podemos combinarlo con otros operadores como +, -.

```
x+=2 #Incrementa x. En Ruby no existe ++
y-=2 #Decrementa y. Tampoco existe --
z*=3
```

A esto que hicimos arriba se le llama asignaciones abreviadas, a continuación te dejo una tabla con varias de ellas.

<i>Asignación abreviada</i>	<i>Significado</i>
$x+=y$	$x = x + y$
$x-=y$	$x = x - y$
$x*=y$	$x = x * y$
$x/=y$	$x = x / y$
$x\%=y$	$x = x \% y$
$x**=y$	$x = x ** y$
$x\&\&=y$	$x = x \&\& y$
$x =y$	$x = x y$
$x\&=y$	$x = x \& y$
$x =y$	$x = x y$
$x^-=y$	$x = x ^ y$
$x<<=y$	$x = x << y$
$x>>=y$	$x = x >> y$

Tabla 2.3: Asignaciones abreviadas

Ruby también soporta asignación paralela. Veamos algunos ejemplos

```
x,y=1,2 #x=1 y=2
x,y=y,x #x=y y=x
a,b,c=[1,"hola",true] #a=1 b="hola" c=true
x=9,8,7 # x=[9,8,7]
```

Si ponemos más variables que valores a asignar, sucede lo siguiente:

```
x,y,z=1,2 # x=1 y=2 z=nil
```

Y si ponemos mas valores que variables:

```
x,y=5,6,7 # x=5 y=6 El 7 es descartado
```

Con la asignación paralela también podemos hacer esto:

```
x,(y,z)=1,2 # x=1 y,z=2
```

Así mismo por la interpretación dinámica, una variable en Ruby puede tener distintos valores y pueden ser objetos de diferentes tipos.

```
x=5
x="cinco"
x= 5.5
```

2.2.2 Constantes

Las constantes son referencias inmutables a un objeto, estas son creadas cuando son asignadas por primera vez. Las constantes en Ruby se deben poner con letras mayúsculas. Si se intenta cambiar su valor en algún momento, el intérprete nos mandará una advertencia, pero no marcará error. Veamos algunos ejemplos:

```
PI=3.14159
E=2.7182
PI=3.1416 #Esto no marca error pero sí genera un warning
```

2.3 True, false and nil

Antes de empezar a trabajar con los condicionales vamos a revisar los valores de *true*, *false* y *nil*. En realidad estos valores son instancias de las clases `TrueClass`, `FalseClass` y `NilClass` respectivamente. No existe la clase booleana en Ruby.

`Nil` se usa para indicar ausencia de valor, es la versión de `null` en Ruby. Debe quedar muy claro que `true` no es 1, que `false` y `nil` no es 0. `Nil` se comporta como `false`. Y cualquier cosa que no sea `nil` o `false` será `true`.

2.4 Operadores de comparación, de igualdad y booleanos

Empecemos con los operadores de comparación. En ruby existen los siguientes:

- ◇ Mayor que `>`
- ◇ Menor que `<`
- ◇ Mayor o igual que `>=`
- ◇ Menor o igual que `<=`
- ◇ Igual que `<=>`

Los primeros cuatro operadores regresan `true` si se cumple la comparación, en caso contrario devuelve falso. El último operador actúa de modo diferente. Si el operando del lado izquierdo es menor que el del lado derecho, regresa -1. Si el operador de la izquierda es más grande que el de la derecha retorna 1. En caso de que ambos sean iguales devuelve 0.

Veamos algunos ejemplos de todos estos operadores.

```
puts 6>5 # true
puts 6<5 # false
puts 6>=5 # true
puts 6<=5 # false
puts 6<=>5 # Regresa 1
```

```
puts 5<=>6 # Regresa -1
puts 5<=>5 # Regresa 0
```

En cuanto a los operadores de igualdad tenemos los siguientes:

- ◇ Igual ==
- ◇ Diferente !=

Estos operadores regresan true si se cumple la igualdad o false en caso contrario. Veamos un ejemplo:

```
a=5
b=5
c=6
puts a==b # true
puts a==c # false
puts a!=b # false
puts b!=c # true
```

Ahora revisemos los operadores booleanos, dentro de los cuales tendremos los siguientes:

- ◇ &&
- ◇ ||
- ◇ !
- ◇ and
- ◇ or
- ◇ not

&& y and realizan la operación booleana AND. || y or realizan la operación OR. ! y not realizan la operación NOT. &&, || y ! tienen mayor precedencia que and, or y not, pero se usan para tener una mayor legibilidad en el código.

2.5 Arreglos

Un arreglo es una secuencia ordenada de valores a la cuál podemos acceder a través de su posición o índice. El primer valor de un arreglo tiene el índice 0. Si nosotros preguntamos la longitud o el tamaño de nuestro arreglo, Ruby nos dará el número de elementos que hay en el arreglo.

Ahora, si nosotros tratamos de acceder a índices negativos, Ruby nos dará los últimos elementos del arreglo. Es decir, si nosotros le pedimos el elemento -1, Ruby nos devolverá el último elemento del arreglo, si le pedimos el -2 nos dará el penúltimo. Si tratamos de acceder a un índice que no existe, Ruby regresará nil.

Los arreglos en Ruby son mutables y también sus elementos pueden ser de distintas clases (tipos). Veamos algunos ejemplos.

#Declaración de arreglos

```
[] #Este es un arreglo vacío
[1,2,3] #Un arreglo con 3 elementos
[1,2.3,"abc",false,[0,5,10]] #Arreglo con distintos tipos
de objetos
```

#Elementos de un arreglo

```
arreglo=[5,8,13,21,[34,55],89]
puts arreglo[0] #Imprime 5
puts arreglo[3] #Imprime 21
puts arreglo[-1] #Imprime 89
puts arreglo[4] #Imprime 34 y 55
puts arreglo[4][0] #Imprime 34
puts arreglo[-7] #Regresa nil, no se puede acceder a un
elemento antes del arreglo
```

Revisemos algunas otras cosas que podemos hacer con los arreglos, como formas de inicializar, de acceder a sus elementos, de agregar y sustituir elementos.

#Algunas otras formas de inicializar arreglos

```
vacio = Array.new() #Creamos un objeto de la clase
arreglo, es equivalente a []
```

```
puroNil = Array.new(2) #Un arreglo de la forma [nil, nil]
```

```
otro = Array.new(2,1) #Un arreglo de esta forma [1,1]
otro2 = Array.new(otro) #Realiza una copia del arreglo
otro
```

```
arreglo=Array.new(4){|x| x+1} #4 elementos con x de 0 a 3
y x+1. Devuelve este arreglo [1,2,3,4]
```

```
a = [1,2,3,4,5]
puts a[a.size-1] #Devuelve el último elemento del arreglo
puts a[-a.size] #Devuelve el primer elemento del arreglo.
puts a[0]="uno" #El arreglo queda a=["uno",2,3,4,5]
```

```
a[-6] = 0 #Esto no se puede, no se puede asignar antes
del primer elemento del arreglo.
```

```
a[7] = 8 #El arreglo queda a=["uno",2,3,4,5,nil,nil,8]
```

En Ruby existen operaciones con arreglos, vamos a revisar.

```
a = [1,2,3]+["cuatro","cinco"] #a=[1,2,3,"cuatro","cinco"]
a = a+[6,[7,8]] #a=[1,2,3,"cuatro","cinco",6,[7,8]]
```



```

a = a+9 #Es un error, solo sirve con arreglos. Así
        tendría que ser a=a+[9]

a = [ 'a', 'b', 'c', 'd', 'e' ] - [ 'a', 'c', 'e' ] #Dara como
        resultado a=['b', 'd']
#Cuando hacemos la resta solo se quitan los elementos
        repetidos

#Para agregar elementos al final del arreglo podemos usar
        el operador <<
a = []
a<<1 #a=[1]
a<<[2,3] #a=[1,[2,3]]

a = [0]*3 #a=[0,0,0]

#También podemos utilizar los operadores & y |
#Estos tratarán a los arreglos como si fueran conjuntos
#Haran las operaciones de unión e intersección
respectivamente.
a = [1,2,2,3,3,3,4,4,4,4]
b = [5,5,5,5,5,6,6,6,6,6,6,2]
a|b #[1,2,3,4,5,6]
b|a #[5,6,2,1,3,4] mismos elementos, diferente orden
a&b #[2]

    Además de todos estos métodos y operaciones con arreglos, existen otros
    que son aún más poderosos y más útiles por ser los más usados. Revisemos este
    último ejemplo.

#Si queremos recorrer todo el arreglo, lo podemos hacer a
        través del método each

a = [1,2,3,4,5]
a.each do |numero|
  puts numero*2
end
#Imprimira el doble de todos los números del arreglo

a.each {|n| puts n*2} #Otra forma

a.length #Devuelve la cantidad de elementos que hay en el
        arreglo
a.first #Devuelve el primer elemento
a.last #Devuelve el último elemento
a.delete(4) #Borra el 4, quedando el arreglo a=[1,2,3,5]

```

```

a.sort #Ordena los elementos del arreglo
#Si queremos que el arreglo se modifique debemos agregar
  un ! al final. Es decir, a.sort!

#Esto es muy util cuando trabajamos con arreglos de
  cadenas
países = %w(Mexico Brasil Argentina Colombia Venezuela)
#Convierte cada palabra a un elemento del arreglo
#países = ["Mexico", "Brasil", "Argentina", "Colombia", "
  Venezuela"]

```

2.6 Símbolos

Los *símbolos* son cadenas internas inmutables en Ruby. Los símbolos no contienen valores como las variables. Un símbolo solamente es una etiqueta, un nombre, no más. Dado un símbolo, siempre se refiere al mismo objeto en todo el programa, esto implica un ahorro de tiempo y memoria.

Entonces, ¿cuándo debe usarse una cadena y cuándo un símbolo? Si el contenido del objeto es importante, se usa una cadena. En cambio si lo importante es la identidad del objeto, entonces usaremos un símbolo.

Deben de ser escritos con dos puntos previos a la cadena (':'). Revisemos algunos ejemplos.

```

puts "hola".object_id
puts "hola".object_id
#Se imprimen diferentes id

puts :algo.object_id
puts :algo.object_id
#Se imprimio lo mismo

a = :simbolo #Objeto de tipo Symbol
b = :"simbolo" #Es el mismo objeto de arriba
puts a==b #Nos imprimira true, ya que son el mismo objeto

c = :'simbolo con espacios'
d = "cadena"
e = :#{d}
puts e #Imprime cadena

#Esta es otra sintaxis para símbolos
%sa #Es lo mismo que :a

```

También se pueden convertir cadenas en símbolos y viceversa. Dos cadenas pueden tener el mismo contenido y sin embargo ser diferentes objetos, como lo vimos en el ejemplo de arriba. En cambio con los símbolos esto no ocurre.

Esto quiere decir que dos cadenas pueden convertirse en el mismo símbolo, pero al convertir dos símbolos en cadenas, serán diferentes objetos. Veamos que métodos se utilizan para lograr esto.

```
x = "cadena"
simbolo = x.intern #Se convierte la cadena en símbolo.
simbolo = x.to_sym #Este es otro método para convertirlo en símbolo
x = simbolo.to_s #Regresamos el símbolo a cadena
x = simbolo.id2name #Otro método para pasar de símbolo a cadena
```

Si se desea saber la igualdad de dos objetos, siempre será más rápido comparar dos símbolos a comparar dos cadenas.

2.7 Hashes

Los *hashes* son conocidos como diccionarios, arreglos asociativos o mapas. Son objetos que tienen una *llave* (*key*) y un *valor*. Veamos como son los hashes en Ruby.

```
frutas = Hash.new #Se crea un nuevo y vacío objeto Hash
frutas["manzana"] = "roja"
#Donde "manzana" es la llave y "rojo" el valor
frutas["piña"] = "amarilla"
puts frutas["piña"] #Imprimirá amarilla

romanos = Hash.new
romanos["X"] = 10
romanos["L"] = 50
puts romanos["X"] + romanos["L"] #Imprime 60
```

2.8 Rangos

Capítulo 3

Estructuras de control y bucles

Revisemos las diferentes estructuras de control y bucles (loops) que podemos usar en Ruby, veremos que en este lenguaje tenemos una gran variedad de opciones que podemos usar para realizar nuestros scripts, algunos son muy poderosos y todo dependerá de lo que queramos realizar.

3.1 Condicionales

Dentro de este rubro contamos con *if*, *unless* y *case*. Así como el operador ternario (*?*). Empecemos con el más básico de ellos. El condicional *if*.

3.1.1 If, else, elsif

Iniciemos con *if*, esta es la forma básica de *if* en Ruby.

```
if expresión
  código
end
```

El código que se encuentra entre *if* y *end* se ejecutará si y solo si al evaluar la expresión no se regresa *false* o *nil*. La expresión debe encontrarse separada del código por alguno de los siguientes tres delimitadores:

- ◇ Un salto de línea
- ◇ Un punto y coma (;)
- ◇ La palabra *then*

Aquí algunos ejemplos.

```
if 6>5
  puts "Seis es mayor que cinco"
end
```

```
if x!=8 then x+=1 end
```

```
if a>b; puts "a menor que b" end
```

```
if x<9 then
  puts "x es menor a 9"
end
```

En Ruby no es necesario poner nuestra expresión entre paréntesis, para eso contamos con los delimitadores antes mencionados. Así mismo sí es necesario poner la palabra `end`.

También podemos anidar `if`.

```
if a>b then
  puts "a es mayor que b"
  if a<c then
    puts "pero a es menor que c"
  end
end
```

Veamos ahora como funciona el *else*. Else se ejecutará si la condición del `if` es falsa. Esta es su estructura básica:

```
if expresión
  código
else
  código
end
```

Cuando usamos `else` ya no necesitamos asociarlo con una expresión, debido a que solo se ejecutará si el `if` no lo hace. También es importante observar que solo usamos un `end`, no se necesita un `end` para el `if` y otro para el `else`. Un solo `end` que engloba a ambos. Ahora un ejemplo.

```
if a>b then
  puts "a es mayor que b"
else
  puts "b es mayor que a"
end
```

Pero que pasa cuando queremos evaluar dos o más condiciones. Bueno para eso en Ruby existe *elsif*. Con `elsif` podemos agregar una o más condiciones que deseamos que sean evaluadas. `Elsif` es como otro `if`, así que de igual manera podemos usar los delimitadores ya antes vistos.

Revisemos la estructura básica si usamos `if`, `elsif` y `else` al mismo tiempo.

```

if expresión1
  código
elsif expresión2
  código
elsif expresión3
  código
...
elsif expresiónN
  código
else
  código
end

```

Revisemos un pequeño ejemplo donde se verifique si *x* es un número positivo o un número negativo o es cero.

```

if x>0 then
  puts "x es un número positivo"
elsif x<0 then
  puts "x es un número negativo"
else
  puts "x es cero"
end

```

En Ruby el *if* es más poderoso, ya que este condicional regresa un valor. El valor que regresa es el de la última expresión de código que fue ejecutado o nil en caso de que no se haya ejecutado nada. Para dejar esto más claro veamos un ejemplo:

#Primera forma

```

if letra=='a' then
  x=1
elsif letra=='b' then
  x=2
elsif letra=='c' then
  x=3
else
  x=4
end

```

#Segunda forma, aprovechando el poder de Ruby

```

x = if letra=='a' then 1
    elsif letra=='b' then 2
    elsif letra=='c' then 3
    else 4
  end

```

If también se puede usar como modificador, de la siguiente forma.

código **if** condición

#Ejemplo

puts x **if** x

#Lo que hace nuestro ejemplo es ver si esta definida la variable x, y de ser así la imprime.

Esta forma es recomendable cuando la condición es trivial o por lo general la condición siempre regresa true.

3.1.2 Case

El *case* es muy similar al if. Sirve para verificar una serie de condiciones y se ejecutará la que cumpla con ella. Veamos su estructura elemental.

```
case
  when expresión1
    código
  when expresión2
    código
  when expresión3
    código
  ...
  when expresiónN
    código
  else
    código
end
```

El else que ves al final puede ir o no, es opcional. Pero dejame decirte que sí importa el orden. Si vas a utilizar else, ese debe ir al final, después de todas las cláusulas *when*.

La forma de actuar del case es la siguiente: revisa condición por condición, una a una de manera secuencial y en el momento que alguna cumple, se ejecuta el código y en dado caso de que hubiera más condiciones, estás ya no son consideradas.

También con el case puedes realizar asignaciones, tal como lo hicimos con el if. Case regresa el último código que ejecuto, en dado caso de que ninguna cláusula when regrese true y no se cuente con un else, el valor que regresará case será nil.

Al igual que con if, puedes usar o no la palabra then. En dado caso de que quieras verificar mas de una condición en un when, puedes utilizar la coma (,) para separar tus condiciones. La coma actuaría como el operador ||.

Veamos algunos ejemplos ya en código.

#Esta es una forma de usar el case

```
case
  when x==1
    puts "uno"
  when x==2
    puts "dos"
end
```

#Otra forma de utilizarlo

```
numero= case x
  when 1 then "uno"
  when 2 then "dos"
end
```

*#También podemos utilizar rangos con el case, los rangos
los revisaremos con mayor profundidad más adelante*

```
case x
  when 1..10 then
    puts "entre 1 y 10"
  when 11..20 then
    puts "entre 11 y 20"
  else
    puts "no considere ese caso"
end
```

#Último dos ejemplos

```
a=case
  when x == 1, x == 0 then "equis es uno o cero"
  when x == 2 || x == 3 then "equis es dos o tres"
end

numero = case x
  when 1
    "one"
  when 2 then "two"
  when 3; "three"
  else "otro"
end
```

La sentencia case utiliza internamente el operador === para verificar las distintas condiciones. Dentro de la naturaleza orientada a objetos de Ruby, === lo interpreta el objeto que aparece en la condición when. Mira el siguiente código:

```
puts case x
  when String then "es cadena"
  when Numeric then "es número"
  when TrueClass, FalseClass then "es booleano"
```



```

    else "es otro"
  end

```

3.1.3 Unless

En Ruby existe una versión contraria al `if` y esa es el *unless*. El `unless` ejecutará código solamente si la expresión evaluada regresa un `false` o un `nil`.

La sintaxis es muy parecida al `if`, aunque es importante mencionar que aquí no existe el `elsif`. Veamos su estructura básica.

```

#Unless básico
unless condición
  código
end

```

```

#Unless con else
unless condición
  código
else
  código
end

```

Ahora un ejemplo para que quede un poco más claro.

```

#Mandaremos un mensaje si en número es negativo
unless x>-1 then
  puts "Ese es un número negativo"
else
  puts "No es un número negativo"
end

```

Como puedes observar es prácticamente lo mismo que el `if`, pero al revés. También puedes usar el `unless` como modificador, igual que lo platicamos en `if`. Solo es cosa de comprender como funciona y utilizar nuestro ingenio de programadores para hacer un buen uso de él.

3.1.4 Operador ternario

El operador ternario es una especie de `if` simplificado y elegante. Hace uso del operador `?`. Revisemos como funciona.

```
condición ? código : código
```

Ahora un ejemplo.

```
puts x>0 ? "Es mayor que cero" : "Es cero o menor"
```

Como dijimos simple y elegante. Una manera más de verificar alguna condición.

3.2 Bucles

3.2.1 While

While sirve para hacer iteraciones mientras la condición que le pongamos sea true, en el momento que deje de serlo termina su ejecución. Esta es su estructura.

```
while condición do
    código
end
```

La palabra *do* es como el then del if. De igual forma puede ir o no, también podemos usar punto y coma o un salto de línea al igual que en if. Es importante mencionar que el while ejecutará el código mientras la condición que hayamos puesto no regrese false o nil. Para lograr esto nosotros tenemos que cuidar que el loop no se vuelva infinito, debemos de tener mucho cuidado con la condición que pongamos y la forma en que haremos que se cumpla dicha condición. Un ejemplo

```
a=0
while a<10 do
    puts a
    a+=1
end
#Imprime del 0 al 9, observemos que nosotros tenemos que
    indicar como irá cambiando a.
```

3.2.2 Until

Until es la versión contraria de while. Ya que la porción de código que esta adentro se ejecuta mientras la condición sea falsa. En el momento en que al evaluar la condición se regrese un true, el bucle se termina y se detiene la ejecución.

```
until condición do
    código
end
```

#Ejemplo

```
b=10
until b==0 do
    puts b
    b-=1
end
#Imprime del 10 al 1
```

3.2.3 For

3.3 Iteradores

3.3.1 Iteradores

3.3.2 Objetos enumerables

3.3.3 Enumeradores

Capítulo 4

Funciones

Capítulo 5

Programación Orientada a Objetos

Capítulo 6

Programación funcional

Capítulo 7

I/O y manejo de Archivos

Capítulo 8

Hilos

Capítulo 9

Miscelánea Ruby

Capítulo 10

Gemas