Aprende a programar

por Chris Pine

Índice

Sobre la versión en español
Un punto de partida para el futuro programador
    Ideas para maestros
    Acerca del tutorial original
    Agradecimientos
0. Preparándose
        Instalación en Windows
        Instalación en Macintosh
        Instalación en sistemas basados en Linux y Unix
1. Números
    Introducción a PUTS
    Enteros y fraccionarios
    Aritmética simple
    Algunas cosas para intentar
2. Letras
    Aritmética de cadenas
    12 vs '12'
    Problemas
3. Variables y asignación
4. Mezclando todo
    Conversiones
    Otra mirada a puts
    Los métodos gets y chomp
    Algunas cosas para intentar
5. Más acerca de los métodos
    Métodos creativos para texto
    Algunas cosas para intentar
6. Matemáticas avanzadas
    Más aritmética
    Números aleatorios
    El objeto Math
7. Control del flujo
    Métodos de comparación
    Ramificación
    Ciclos
    Un poco de lógica
    Algunas cosas para intentar

8. Arreglos e iteradores
    El método each
    Más métodos de arreglos
    Algunas cosas para intentar
9. Escribiendo tus propios métodos
    Parámetros en los métodos
    Variables locales
    Retorno de valores
    Un ejemplo aún más grande
    Algunas cosas para intentar
10. Clases
    La clase Time
    Algunas cosas para intentar
    La clase Hash
    Extendiendo clases
    Creando clases
    Variables de instancia
    Algunas cosas para intentar
11. Bloques y procs
    Métodos que reciben procs
    Métodos que devuelven procs
    Pasando bloques (no procs) a métodos
    Algunas cosas para intentar
12. Más allá de éste tutorial
    Recursos en español
    Tim Toady
    El fin
13. Soluciones a los problemas propuestos

Sobre la versión en español

Hace algunos años, me enteré de la existencia del lenguaje de programación Ruby y sus bondades enfocadas a ofrecer la posibilidad de hacer disfrutable la actividad de escribir código en desarrollo de programas de ordenador. Para personas hemos pasado por la experiencia de lo obscuro que puede llegar a ser el código en C/C++ y la excesiva cantidad de código que puede requerirse en Java, el pragmatismo y la simplicidad mostradas en los ejemplos de Ruby son una bocanada de aire fresco. Esto me llevó a buscar un curso introductorio para aprender la sintaxis y los conceptos básicos de lenguaje, además de ponerme a practicar.

La creciente y activa comunidad en torno a Ruby ha generado bastante información al respecto, así que conseguir información en la Internet no es problema. Pero tuve la fortuna de toparme con el tutorial de Chris Pine, que da una introducción bastante accesible a la programación en general, mediante sencillas explicaciones e interesantes ejercicios que a la vez muestran las características propias del lenguaje Ruby; llevando al lector a recorrer los temas de los números y operaciones aritméticas, las cadenas de caracteres, las variables y la operación de asignación, los métodos, el control del flujo de los programas, los arreglos y las formas de recorrerlos, las clases y los bloques de instrucciones, para finalizar con consejos para continuar con el aprendizaje más allá de éste tutorial.

Es por eso que me di la tarea de solicitar al sr. Pine su consentimiento para traducir y publicar su material en español, con el propósito de ofrecer su excelente contenido a los hispanohablantes en su lengua. Si llegaste hasta éste documento con la intención de iniciar el aprendizaje programación, seguramente encontrarás que está escrito de forma amena y que mantiene la atención (cualidades que espero retener lo más posible al traducirlo). De igual forma, si conoces a alguien interesado, no dudes en compartirlo con esa persona. Seguramente le será de utilidad.

Si encuentras algún error en la traducción o crees que se podría mejorar algún párrafo o frase para hacerlo más fácil de entender sin perder el sentido, puedes contactarme en ésta dirección, o bien, colaborar con el proyecto disponible en Github.

-- David O' Rojo

Un punto de partida para el futuro programador

Creo que todo esto comenzó tiempo atrás, en el 2002. Estaba pensando acerca de enseñar programación y que gran lenguaje sería Ruby para aprender cómo programar. Quiero decir, nos encontrábamos todo excitados acerca de Ruby porque era poderoso, elegante y, de verdad, simplemente divertido, pero me parecía que también sería una gran manera de adentrarse a la programación en primer lugar.

Desafortunadamente, no había mucha documentación orientada a principiantes en aquel entonces. En la comunidad, algunos de nosotros hablábamos sobre lo que un tutorial de «Ruby para el novato» necesitaría y, de forma más general, como enseñar a programar. Mientras más pensaba en eso, más tenía que decir al respecto (lo cual me sorprendió un poco). Finalmente alguien dijo, «Chris, ¿por qué no escribes un tutorial en lugar de hablar de eso?». Así que lo hice.

Y el resultado fue muy bueno. Tenía todas esas ideas que eran buenas *en teoría*, pero la tarea de hacer un gran tutorial para no-programadores tenía más reto de lo que había imaginado. (Quiero decir, parecía bueno para mí, pero yo ya sabía cómo programar).

Lo que me salvó fue que hice realmente fácil para las personas el contactarme y siempre traté de ayudarles cuando se encontraban dificultades. Cuando veía que muchas personas tenían problemas en algún lugar, lo reescribía. Fue mucho trabajo, pero poco a poco se volvió mejor y mejor.

Un par de años después, se estaba volviendo bastante bueno. :-) Muy bueno, de hecho, estaba listo para pronunciarlo terminado y moverme a algo más. Y justo en ese momento surgió la oportunidad de convertir el tutorial en un libro. Como prácticamente ya estaba hecho, se me figuro que no habría problema. Solo tendría que limpiar algunos puntos, añadir más ejercicios, tal vez algunos ejemplos más, unos cuantos capítulos más, pasarlo a 50 revisiones más...

Me tomó otro año, pero ahora creo que es realmente bueno, sobre todo debido a los cientos de valientes almas que me ayudaron a escribirlo.

Lo que está en este sitio es el tutorial original, más o menos sin cambios desde el 2004. Para leer la versión mejorada y más amplia, seguramente te gustaría revisar el libro.

Ideas para maestros

Hubo algunos pocos principios a los que traté de adherirme. Creo que hicieron el proceso de aprendizaje mucho más fluido; aprender a programar ya es suficientemente difícil. Si estás enseñando o guiando a alguien sobre el camino hacia el reino del hacking, estas ideas podrían ayudarte también.

Primero, traté de separar los conceptos tanto como fuera posible, así el estudiante sólo tendrá que aprender un concepto a la vez. Esto fue difícil al principio, pero muy fácil cuando tomé un poco de práctica. Algunas cosas deben ser enseñadas antes que otras, pero me sorprendió cuán poca precedencia jerárquica realmente hay. Eventualmente sólo tuve que escoger un orden y arreglé las cosas de forma que cada nueva sección fuera motivada por las precedentes.

Otro principio que he mantenido en mente es enseñar sólo una forma de hacer algo. Es un beneficio obvio en un tutorial para personas que nunca han programado antes, por ésta razón: una manera de hacer las cosas es más fácil de aprender que dos. Tal vez un beneficio más importante, sin embargo, es que entre menos cosas le enseñes a un nuevo programador, más creativo y listo debe ser en sus soluciones. Como mucho de la programación es resolver problemas, es crucial promover esas habilidades tanto como sea posible en cada paso.

Traté de colocar nuevos conceptos de programación sobre los conceptos que un nuevo programador ya tiene; de presentar ideas de tal forma que su intuición llevará la carga más que el tutorial. La programación orientada a objetos se presta muy bien para esto. Fui capaz de comenzar a referirme a «objetos» y «diferentes tipos de objetos» muy temprano en el tutorial, deslizando esas palabras en el más inocente de los momentos. No decía cosas como «todo en Ruby es un objeto» o «números y cadenas de caracteres son objetos» porque estas frases no significan nada para un nuevo programador. En vez de eso, yo hablaría de cadenas de caracteres (no «objetos cadena»), y algunas veces me referiría a objetos simplemente diciendo «las cosas en esos programas». El hecho de que todas esas *cosas* en Ruby *son* objetos hizo que este tipo de estrategias de mi parte funcionará muy bien.

Aunque yo deseaba evitar el uso de jerga orientada a objetos innecesaria, quería estar seguro de que si ellos necesitaban aprender una palabra aprendieran la correcta. (No quiero que ellos tengan que aprender dos veces, ¿verdad?). Así que les llame «cadenas de caracteres» no «texto». Los métodos necesitan ser llamados de alguna forma así que los llame «métodos».

En cuanto a los ejercicios, creo que ideé unos muy buenos, pero nunca tienes

demasiados. Honestamente, apostaría que gaste la mitad de mi tiempo tratando de formular ejercicios divertidos e interesantes. Ejercicios aburridos matarán absolutamente cualquier deseo de programar, mientras que el ejercicio perfecto crea esa comezón que el nuevo programador no puede más que rascar. Resumiendo, el tiempo gastado en formular buenos ejercicios no es demasiado.

Acerca del tutorial original

Las páginas de éste tutorial (aun ésta), son generadas por un programa en Ruby, claro está. :-) Todos los códigos de ejemplo son automáticamente ejecutados y sus resultados son mostrados como han sido generados. Creo que es la mejor, más fácil y genial manera de asegurarme que todo el código que presento funciona exactamente como yo digo que lo hace. No tienes que preocuparte por el que yo haya copiado mal la salida de alguno de los ejemplos, o no haya probado algún código; todo ha sido probado.

Agradecimientos

Finalmente, quiero agradecer a todos en la lista de correo ruby-talk por sus comentarios y el ánimo que me dieron. A todos los magníficos correctores que me ayudaron a hacer el libro mucho mejor de lo que pudiera haber hecho yo solo. A mi querida esposa, especialmente por ser mi principal corrector-conejillo-de-indias-musa. A Matz por crear éste fabuloso lenguaje, y a los Programadores Pragmáticos por decirme acerca de él – ¡Y por supuesto publicar mi libro!

Si notas algún error, tanto en un programa como en el texto, o tienes algún comentario o sugerencia o buenos ejercicios que pudiera incluir, por favor déjame saber[1] .

[1] Es probable que el sr. Chris Pine prefiera recibir correos en el idioma inglés. Si deseas enviar algún comentario sobre la versión en español, será mejor que escribas a ésta dirección.

0. Preparándose

Cuando programas una computadora, tienes que «hablar» en un lenguaje que tu computadora entiende: un lenguaje de programación. Hay montones y montones de diferentes lenguajes de programación ahí afuera; muchos de ellos son excelentes. En éste tutorial elegí usar mi lenguaje de programación favorito, Ruby.

Además de ser mi favorito, Ruby es también el lenguaje de programación más sencillo que haya visto (y he visto bastantes). De hecho, ésta es la verdadera razón de escribir el tutorial: no decidí escribir un tutorial y luego escogí Ruby porque es mi favorito; en vez de eso, encontré a Ruby tan fácil de usar que decidí el que tendría que un buen tutorial para los principiantes. Es la simplicidad de Ruby lo que promovió éste tutorial, no el hecho de que sea mi favorito. (Escribir un tutorial similar usando otro lenguaje como C++ o Java, hubiera requerido cientos y cientos de páginas). Pero no pienses que Ruby es un lenguaje para principiantes sólo porque es fácil. Es un poderoso lenguaje de programación de uso profesional (si es que ha habido alguno).

Cuando escribes algo en un lenguaje humano, lo que escribes es llamado texto. Cuando escribes algo en un lenguaje de computadora, lo que escribes es llamado código. He incluido muchos ejemplos de código en Ruby a través de éste tutorial, la mayoría de ellos son programas completos que puedes ejecutar en tu computadora. Para hacer el código más fácil de leer, he coloreado las partes en código con diferentes colores. (Por ejemplo, los números siempre son `verdes`). Cualquier cosa que necesites escribir estará en una caja gris, y cualquier cosa que un programa imprima estará en una caja azul.

```
Caja gris: El código que escribes.
```

```
Caja azul: Lo que la computadora presenta en pantalla.
```

Si te cruzas con algo que no entiendes o si tienes una pregunta que no ha sido respondida, toma nota y continúa leyendo. Es muy posible que la respuesta aparezca en un capítulo más adelante. Sin embargo, si tu pregunta no fue respondida ya, en el último capítulo te diré donde puedes preguntar. Hay muchas maravillosas personas allá afuera deseando ayudar; sólo necesitas saber dónde están.

Pero primero necesitamos descargar e instalar Ruby en tu computadora.[2]

[2] Puedes encontrar otras formas recomendadas y alternativas de instalar (o actualizar) Ruby en tu computadora en la siguiente dirección: http://www.ruby-lang.org/es/downloads/

Instalación en Windows

La instalación de Ruby en Windows es fácil. Primero necesitas descargar Ruby Installer. Podría haber un par de versiones de las cuales escoger; éste tutorial usará la versión 1.8.4, así que asegúrate de que tu descarga sea una versión superior o al menos igual. (Yo obtendría la versión más reciente disponible). Después, simplemente inicia el programa de instalación. Éste te preguntará donde quieres instalar Ruby. A menos que tengas una buena razón para cambiar el lugar, deja la ruta por defecto.

Al programar necesitarás ser capaz de escribir y ejecutar los programas. Para hacer esto necesitarás un editor de textos y la línea de comandos.

En la red se encuentra el excelente editor de textos Notepad++, que colorea la sintaxis de muchos lenguajes, incluyendo Ruby (y que además es gratuito).

También es una buena idea crear una carpeta en algún lugar para guardar todos tus programas. Asegúrate de que cuando guardes un programa lo hagas en esa carpeta.

Para usar la línea de comandos, simplemente navega a la carpeta donde guardarás tus programas. Presiona `Alt + D` para seleccionar la barra de direcciones, escribe `cmd` y presiona `Intro` para mostrar la línea de comandos. Dentro de la línea de comandos, escribiendo `cd ..` subirás un nivel de carpeta, y `cd nombre_carpeta` te pondrá dentro de la carpeta llamada `nombre_carpeta`. Para ver todos las carpetas dentro de la carpeta donde en encuentras actualmente, escribe `dir /ad`.

¡Y eso es todo! Estás listo para aprender a programar.

Instalación en Macintosh

Si tienes Mac OS X 10.2 (Jaguar) o superior, ¡entonces ya tienes Ruby en tu sistema! ¿Qué podría ser más fácil? Infortunadamente, no creo que puedas usar Ruby en Mac OS X 10.1 y versiones anteriores.

Al programar, necesitas ser capaz de escribir código. Para hacer esto, necesitarás un editor de textos y la línea de comandos. Puedes acceder a tu línea de comandos a través de la aplicación `Terminal` (encontrado en `Aplicaciones/Utilidades`).

Como editor de texto, puedes usar cualquiera que te sea familiar o con el que estés confortable. Si usas `TextEdit`, asegúrate de guardar tus programas como solo texto, de otra forma tus programas no funcionarán. Otras opciones para programar son `emacs`, `vi` y `pico`, la cuales son todas accesibles desde la línea de comandos.

¡Y eso es todo! Estás listo para comenzar a programar.

Instalación en sistemas basados en Linux y Unix

Las distribuciones recientes de sistemas basados en Linux permiten la instalación de Ruby de una manera muy fácil a través de sus administradores de paquetes. Pero antes de eso, debes cerciorarte que no hay una versión instalada en tu sistema que puedas utilizar.

Simplemente escribiendo el comando `ruby -v` en una terminal, se desplegará un mensaje parecido a `ruby 2.0.0p247 (2013-06-27 revision 41674) [i686-linux]`. De no ser así, instala el intérprete de Ruby con el comando correspondiente a tu distribución:

- Ubuntu/Mint/Debian: `sudo apt-get install ruby`
- Fedora/Red Hat: `sudo yum install ruby`
- Arch: `sudo pacman -S ruby`
- Gentoo/Sabayon/Funtoo: `emerge ruby`
- OpenIndiana: `pkg install runtime/ruby-18`

En la mayoría de las distribuciones de Linux puedes utilizar gedit o leafpad para escribir tus programas, además de que seguramente están disponibles vi, vim, pico o nano desde la línea de comandos de casi todas los sistemas basados en Unix.

¡Y eso es todo! Estás listo para aprender a programar.[3]

[3] También está disponible en la red un editor para Linux, con características más avanzadas, llamado Geany.

1. Números

Ahora que está todo preparado, ¡vamos a escribir un programa! Abre tu editor de textos favorito y teclea lo siguiente[4]:

```
puts 1 + 2
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Guarda tu programa (¡sí, eso es un programa!) como `operaciones.rb` (el **.rb** es lo que usualmente ponemos al final de los programas escritos en Ruby). Ahora ejecuta tu programa tecleando `ruby operaciones.rb` en la línea de comandos. Éste debe poner un `3` en tu pantalla. ¿Ves? Programar no es tan difícil.

Introducción a `puts`

¿Qué está pasando en ese programa? Estoy seguro de que puedes adivinar lo que hace `1+2`; nuestro programa es básicamente lo mismo que

```
puts 3
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

`puts` simplemente *pone* en la pantalla cualquier cosa que tenga enfrente.

Enteros y fraccionarios

En la mayoría de los lenguajes de programación (y Ruby no es la excepción) los números sin punto decimal son llamados *enteros*, y los números con punto decimal son usualmente llamados números de *punto flotante*, *coma flotante*, o simplemente, *fraccionarios*.

Aquí hay algunos enteros:

```
5
-295
999999999999999999999
0
```

Y aquí hay algunos fraccionarios:

```
54.321
0.001
-205.3884
0.0
```

[4] Para evitar conflictos de «codificación», es recomendable que agregues la línea `# encoding: UTF-8` al principio de todos los archivos de código donde utilices caracteres acentuados y/o la letra ñ. Si estás usando la versión 2.0 de Ruby o una versión superior, ya no es necesario agregar esa línea.

En la práctica, la mayoría de los programas no usan números fraccionarios, sólo enteros. (Después de todo, nadie quiere ver 7.4 correos, o navegar 1.8 páginas, o escuchar 5.24 de sus canciones favoritas...) Los números fraccionarios son usados para propósitos académicos (experimentos de física y parecidos) y para gráficos en 3D. Aún la mayoría de los programas de dinero sólo manejan enteros; simplemente llevan rastro de los centavos.

Aritmética simple

Hasta ahora podemos hacer lo mismo que una calculadora simple. (Las calculadoras siempre usan números fraccionarios, así que si quieres que tu computadora funcione justo como una calculadora, deberías usar fraccionarios). Para la adición y substracción, usamos `+` y `-`, como ya has visto. Para la multiplicación, usamos `*`, y para la división usamos `/`.

La mayoría de los teclados tienen esas teclas en el teclado numérico en la orilla del lado derecho. Si tienes un teclado más pequeño o una laptop, usualmente podrás acceder a esos símbolos con una tecla especial más la tecla que tiene el símbolo marcado. Ahora intentemos expandir nuestro programa **`operaciones.rb`** un poco. Teclea lo siguiente y después ejecútalo:

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

Esto es lo que el programa retorna:

```
3.0
6.0
-3.0
4.5
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

(Los espacios en el programa no son importantes, sólo hacen el código más fácil de leer). Bueno, eso no fue muy sorprendente. Ahora hay que intentarlo con enteros:

```
puts 1 + 2
```

```
puts 2 * 3
puts 5 - 8
puts 9 / 2
```

```
3
6
-3
4
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Casi lo mismo, ¿verdad? Uh... ¡excepto por ese último! Porque cuando haces aritmética con enteros, obtendrás respuestas con enteros.

Cuando tu computadora no puede obtener la respuesta «correcta», siempre redondea hacia abajo. (Por supuesto, `4` es la respuesta correcta en aritmética de enteros para `9/2`; simplemente es posible que no es la respuesta que estabas esperando).

Puede ser que te preguntes para qué sirve la división de enteros. Bueno, digamos que irás al cine, pero sólo tienes $9. Aquí en Portland, puedes ver una película en el cinema Bagdag por 2 dólares. ¿Cuántas películas puedes ver ahí? `9/2`... `4` películas. 4.5 definitivamente *no* es la respuesta correcta en este caso; no te dejarán ver la mitad de una película, ni dejarán que la mitad de ti entre a ver una película completa... algunas cosas simplemente no son divisibles.

```
puts 5 * (12 - 8) + -15
puts 98 + (59872 / (13 * 8)) * -52
```

```
5
-29802
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Algunas cosas para intentar

Escribe un programa que te diga:

- ¿Cuántas horas hay en un año?

- ¿Cuántos minutos hay en una década?
- ¿Cuántos segundos tienes de edad?
- ¿Cuántos chocolates esperas comer en tu vida? ***Advertencia**: ¡Esta parte del programa puede tardar un poco en calcularse!*

Una pregunta más difícil

- ¿Si tengo 1031 millones de segundos de edad, qué edad tengo en años?

Cuando hayas terminado de jugar con los números, demos un vistazo a las letras.

2. Letras

Así que hemos aprendido todo sobre los números pero, ¿qué hay de acerca de las letras? ¿Palabras? ¿Texto?

Nos referimos a grupos de letras en un programa como *cadenas de caracteres* –puedes pensar en letras impresas que están juntas en un *banner*[5]—. Para hacer más fácil el ver que parte del código es una cadena, las colorearé de **rojo**.

Aquí hay algunas cadenas[6]:

```
'Hola.'

'¡Ruby rockea!'

'El 5 es mi número favorito... ¿cuál es el tuyo?'

'Snoopy dice #%^&*@! cuando se golpea un pie.'

'          '

''
```

Como puedes ver, las cadenas pueden tener signos de puntuación, dígitos, símbolos y espacios en ellas... más que sólo letras. La última cadena no tiene nada dentro, así que la llamamos *cadena vacía*.

Hemos usado **puts** para imprimir números; intentemos ahora con algunas cadenas:

```
puts '¡Hola mundo!'
puts ' '
puts 'Adiós.'
```

```
¡Hola mundo!

```

[5] Banderola.
[6] Hay que señalar que dentro del código de los programas sólo se utilizan la prima (') y la doble prima (") en lugar de los apóstrofos (‘ y “”) tanto para indicar el inicio y como para el cierre de una cadena de texto. Los editores de texto plano y otros programas enfocados a la programación las utilizan por defecto.

```
Adiós.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Eso salió bien. Ahora intenta tus propias cadenas.

Aritmética de cadenas

Justo cómo puedes realizar operaciones aritméticas en números, ¡también puedes hacerlo en cadenas! Bueno, algo así... de cualquier manera, puedes sumar cadenas. Intentemos sumar dos cadenas y veamos que hace **`puts`** con eso.

```
puts 'Me gusta' + 'la tarta de manzana.'
```

```
Me gustala tarta de manzana.
```

¡Uuups! Olvide colocar un espacio entre **`'Me gusta'`** y **`'la tarta de manzana.'`** Los espacios usualmente no importan, pero lo hacen dentro de las cadenas. (Es cierto lo que dicen: las computadoras no hacen lo que *quieres* que hagan, sólo hacen lo que les *dices* que hagan.) Intentemos eso otra vez:

```
puts 'Me gusta ' + 'la tarta de manzana.'
puts 'Me gusta' + ' la tarta de manzana.'
```

```
Me gusta la tarta de manzana.
Me gusta la tarta de manzana.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

(Como puedes ver, no importó a que cadena agregué el espacio.)

Así que puedes sumar cadenas, ¡pero también puedes multiplicarlas! (Aunque, por un número). Observa esto:

```
puts 'parpadea ' * 4
```

```
La computadora te hace ojitos.
```

Sólo bromeo... en realidad hace esto:

```
parpadea parpadea parpadea parpadea
```

Si lo piensas, tiene sentido. Después de todo, `7 * 3` realmente sólo significa `7 + 7 + 7`, por lo que `'moo' * 3` sólo significa `'moo' + 'moo' + 'moo'`.

12 vs '12'

Antes de que vayamos más lejos, debemos asegurarnos de que entendemos la diferencia entre *números* y *dígitos*. `12` es un número, pero `'12'` es una cadena de dos dígitos. Juguemos con esto un poco:

```
puts 12 + 12
puts '12' + '12'
puts '12 + 12'
```

```
24
1212
12 + 12
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Ahora esto:

```
puts 2 * 5
puts '2' * 5
puts '2 * 5'
```

```
10
22222
2 * 5
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Los ejemplos son bastante directos. Sin embargo, si no tienes cuidado en como mezclas tus cadenas y números puede que te metas en...

Problemas

En éste punto, puedes haber intentado algunas cosas que *no* funcionaron. Si no, aquí hay una pequeña muestra:

```
puts '12' + 12
puts '2' * '5'
```

```
#<TypeError: can't convert Fixnum into string>[7]
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Mmm... un mensaje de error. El problema es que realmente no puedes sumar un número a una cadena, o multiplicar una cadena por otra cadena. No tiene más sentido que hacer esto:

```
puts 'Beatriz' + 12
puts 'Alfredo' * 'Juan'
```

Algo más que debes tomar en cuenta: puedes escribir 'puerco' * 5 en un programa, dado que eso sólo significa 5 conjuntos de la cadena 'puerco' sumados juntos. Sin embargo, *no puedes* escribir 5 * 'puerco', ya que eso significa 'puerco' conjuntos del número 5, lo que es un poco tonto.

Finalmente, ¿qué tal si quisieras imprimir '2° 57' 32"'? Podemos intentar esto:

```
puts '2° 57' 32"'
```

Bueno, *eso* no funcionará; yo ni siquiera intentaría ejecutarlo. La computadora pensaría que hemos terminado con la cadena al encontrar el segundo apóstrofo. (Por eso es bueno tener un editor de textos que *colorea la sintaxis* por ti). Así que, ¿cómo dejamos saber a la computadora que queremos continuar con la cadena? Tenemos que *escapar* el apóstrofo, así:

```
puts '2° 57\' 32"'
```

[7] El mensaje de error dice de forma literal que no se puede convertir un número en una cadena. Se requiere de una operación especial para poder realizarlo.

```
2° 57' 32"
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

La diagonal invertida es el carácter de escape. En otras palabras, si tienes una diagonal invertida y otro carácter, a veces son transformados en un nuevo carácter. Sin embargo, la diagonal invertida sólo escapa los apostrofes y a sí misma. (Si piensas acerca de ello, los caracteres de escape siempre deben escaparse a sí mismos). Aquí hay algunos pocos ejemplos en orden (creo):

```
puts 'diagonal invertida al final de una cadena: \\'
puts 'arriba\\abajo'
puts 'arriba\abajo'
```

```
diagonal invertida al final de una cadena: \
arriba\abajo
arriba\abajo
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Como la diagonal invertida *no* escapa una «a», pero *sí* se escapa a sí misma, las dos últimas cadenas son idénticas. No parecen lo mismo en código, pero para tu computadora realmente son lo mismo.

Si tienes otras preguntas, ¡sigue leyendo! Después de todo, no puedo responder todas dudas sólo en *ésta* página.

3. Variables y asignación

Hasta ahora, cuando mandamos imprimir una cadena o número, aquello que se imprime desaparece. Lo que quiero decir es, si quisiéramos imprimir algo dos veces, tendríamos que escribirlos dos veces.

```
puts '...puedes decirlo de nuevo...'
puts '...puedes decirlo de nuevo...'
```

```
...puedes decirlo de nuevo...
...puedes decirlo de nuevo...
```

Sería bueno que pudiéramos escribirlo sólo una vez y después conservarlo... guardarlo en algún lugar. Bueno, podemos, por supuesto – de otra forma, ¡no lo habría mencionado!

Para almacenar una cadena en la memoria de tu computadora, necesitamos darle un nombre a la cadena. Los programadores frecuentemente se refieren a éste proceso como *asignación*, y llaman *variables* a los nombres. Una variable puede ser casi cualquier secuencia de caracteres y números, pero el primer carácter debe ser una letra minúscula.

Intentemos de nuevo el último problema, pero ésta vez le daré a la cadena el nombre **`miCadena`**. (Aunque podría sólo haberla llamado **`cdn`** o **`miPequeñaCadena`** o **`enriqueOctavo`**.)

```
miCadena = '...puedes decir eso de nuevo...'
puts miCadena
puts miCadena
```

```
...puedes decirlo de nuevo...
...puedes decirlo de nuevo...
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Cuando intentes hacer algo a **`miCadena`**, el programa lo hará a **`'...puedes decirlo de nuevo...'`** en su lugar. Puedes pensar en la variable **`miCadena`** como

«apuntando a» la cadena `'...puedes decirlo de nuevo...'`.

Aquí hay un ejemplo un poco más interesante:

```
nombre = 'Yehuda Katz Yukihiro Matsumoto Linus Torvals Alan Turing'
puts 'Mi nombre es ' + nombre + '.'
puts '¡Oh! ¡' + nombre + ' es un nombre realmente largo!'
```

```
Mi nombre es Yehuda Katz Yukihiro Matsumoto Linus Torvals Alan Turing.
¡Oh! ¡Yehuda Katz Yukihiro Matsumoto Linus Torvals Alan Turing es un nombre realmente largo![8]
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

También, así como podemos *asignar* un objeto a una variable, podemos *reasignar* un objeto diferente a esa variable. (Esto es por lo que las llamamos variables: debido a que lo que apuntan puede variar.)

```
compositor = 'Mozart'
puts compositor + ' era «la bomba», es sus días.'

compositor = 'Beethoven'
puts 'Pero, personalmente, yo prefiero a ' + compositor + '.'
```

```
Mozart era «la bomba», es sus días.
Pero, personalmente, yo prefiero a Beethoven.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Por supuesto, las variables pueden apuntar a cualquier clase de objeto, no sólo cadenas:

```
var = 'sólo otra ' + 'cadena'
puts var
```

[8] Y es un nombre totalmente ficticio, compuesto de los nombres de prominentes personajes en el campo de la computación y la industria del software. (El nombre fue cambiado del que se encuentra en el texto original).

```
var = 5 * (1 + 2)
puts var
```

```
sólo otra cadena
15
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

De hecho, las variables pueden apuntar a casi todo... excepto otras variables. ¿Qué pasa si lo intentamos?

```
var1 = 8
var2 = var1
puts var1
puts var2

puts ''

var1 = 'ocho'
puts var1
puts var2
```

```
8
8

ocho
8
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Primero, cuando intentamos apuntar **`var2`** a **`var1`**, en realidad apunto a **`8`** en su lugar (justo como lo hacía **`var1`**). Entonces hicimos a **`var1`** apuntar a **`'ocho'`**, pero como **`var2`** no estaba en realidad apuntando a **`var1`**, **`var2`** se quedó apuntado a **`8`**.

¡Ahora que entendimos las variables, los números y las cadenas de caracteres, aprendamos cómo mezclarlo todo!

4. Mezclando todo

Hemos observado algunos tipos de objetos diferentes (números y letras), hicimos que algunas variables apuntaran a ellos; lo siguiente que queremos hacer es ponerlos a jugar bonito a todos juntos.

Vimos que si queremos que un programa imprima `25`, lo siguiente ***no*** *funciona*, porque no podemos sumar números y cadenas:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Parte del problema es que tu computadora no sabe si tú estás intentando obtener `7(2 + 5)`, o si estás intentando obtener `25('2' + '5')`.

Antes de que podamos sumarlos, necesitamos una forma de obtener la versión en cadena de caracteres de **`var1`**, u obtener la versión en número entero de **`var2`**.

Conversiones

Para obtener la versión en cadena de caracteres de un objeto simplemente le agregamos **`.to_s`** al final:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

```
25
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

De forma similar, **`to_i`** da la versión en número entero de un objeto, y **`to_f`** da la versión en número flotante. Veamos un poco más de cerca lo que estos tres métodos hacen (y *no* hacen):

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Observa que aún después de que obtuvimos la versión en cadena de caracteres de **`var1`** llamando **`to_s`**, **`var1`** continúa apuntando a `2` y nunca a `'2'`. A menos que nosotros específicamente reasignemos **`var1`** (lo que requiere el usar un signo **`=`**), ésta apuntará a `2` por el resto del programa.

Ahora intentemos algunas conversiones más interesantes (y algunas extrañas):

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 es mi número favorito'.to_i
puts '¿Quien te pregunto?'.to_i
puts 'Tu hermana lo hizo.'.to_f
puts ''
puts 'cadenita'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

cadenita
```

```
3
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Esto probablemente te dio algunas sorpresas. El primer resultado es muy común, imprime `15.0`. Después de eso, convertimos la cadena `'99.999'` a su versión en número flotante y entero. El flotante se apareció como esperábamos, mientras que el entero fue, como siempre, redondeado hacia abajo.

En la siguiente parte, tenemos algunos ejemplos de cadenas un poco... *inusuales* siendo convertidas en números. `to_i` ignora lo primero que no entiende y el resto de la cadena desde ese punto en adelante. Así que la primera fue convertida en `5`, pero las otras, como comienzan con letras, fueron ignoradas completamente... por lo que la computadora simplemente eligió cero.

Finalmente, vimos que las dos últimas conversiones no hicieron absolutamente nada, justo como lo esperaríamos.

Otra mirada a `puts`

Hay algo extraño acerca de nuestro método favorito... Observa esto:

```
puts 20
puts 20.to_s
puts '20'
```

```
20
20
20
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¿Por qué estos tres imprimen la misma cosa? Pues, los últimos dos deberían, ya que `20.to_s` es `'20'` Pero, ¿qué hay del primero, *el entero* `20`? Para esto, ¿qué significa escribir *el entero* `20`? Cuando escribes un *2* y luego un *0* en un pedazo de papel, estás escribiendo una cadena de caracteres, no un entero. *El entero* `20` es el número de dedos que tengo en manos y pies; no es un `2` seguido de un `0`.

Bueno, aquí está el gran secreto detrás de nuestro amigo `puts`: antes de que `puts` intente escribir un objeto, usa `to_s` para obtener la versión en cadena de

caracteres de ese objeto. De hecho, la *s* en **puts** se refiere a *cadena*[9]; **puts** realmente significa *colocar cadena*.

Esto no puede parecer muy emocionante ahora, pero hay muchos, *muchos* tipos de objetos en Ruby (¡hasta aprenderás a crear los tuyos!), y es bueno conocer que pasará si intentas usar **puts** en un objeto realmente raro, como una fotografía de la abuela, un archivo de música o algo. Pero eso vendrá después...

Mientras tanto, hay un par de métodos más para ti, y ellos te permiten escribir todo tipo de programas divertidos...

Los métodos `gets` y `chomp`

Si **puts** significa *colocar cadena*, estoy seguro de que adivinas que significa **gets**. Y justo como **puts** siempre *escupe* cadenas de caracteres, **gets** sólo las *atrapa*. Y, ¿de dónde las obtiene?

¡De ti! Bueno, de tu teclado, al menos. Como tu teclado sólo produce cadenas de caracteres, funciona de maravilla. Lo que realmente pasa es que **gets** simplemente se sienta ahí, leyendo lo que escribes hasta que presionas **Intro**.

Vamos a intentarlo:

```
puts gets
```

```
¿Hay eco aquí?
¿Hay eco aquí?
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Claro, cualquier cosas que escribas te será simplemente repetida de regreso. Ejecuta el programa algunas veces e intenta escribir diferentes cosas.

¡Ahora podemos hacer programas interactivos! En éste, escribe tu nombre y te saludará:

```
puts 'Hola... y, ¿cuál es tu nombre?'
nombre = gets
puts '¿Tu nombre es ' + nombre + '? ¡Qué nombre tan hermoso!'
```

[9] De la palabra en idioma inglés, *string*.

```
puts 'Gusto en conocerte ' + nombre + '.   :)'
```

¡Ah! Justo lo ejecute —Escribí mi nombre, y esto es lo que pasó:

```
Hola... y, ¿cuál es tu nombre?
Chris
¿Tu nombre es Chris
? ¡Qué nombre tan hermoso!
Gusto en conocerte, Chris
.   :)
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Hmmm... parece que cuando teclee las letras `C`, `h`, `r`, `i`, `s`, *y* entonces presione **`Intro`**, ¡**`gets`** tomó todas las letras de mi nombre y el **`Intro`**! Por suerte, hay un método justo para este tipo de cosas: **`chomp`**. El cual remueve cualquier **`Intro`** que esté colgando al final de tu cadena de caracteres. Intentemos el programa de nuevo, pero con **`chomp`** para ayudarnos ésta vez:

```
puts 'Hola... y, ¿cuál es tu nombre?'
nombre = gets.chomp
puts '¿Tu nombre es ' + nombre + '? ¡Qué nombre tan hermoso!'
puts 'Gusto en conocerte ' + nombre + '.   :)'
```

```
Hola... y, ¿cúal es tu nombre?
Chris
¿Tu nombre es Chris? ¡Qué nombre tan hermoso!
Gusto en conocerte, Chris.   :)
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Mucho mejor! Nota que como **`nombre`** está apuntando a **`gets.chomp`**, nunca tenemos que decir **`nombre.chomp`**: el nombre ya ha sido *masticado*.[10]

Algunas cosas para intentar

- Escribe un programa que pregunte por el nombre y apellidos de una persona. Al

[10] El verbo en inglés «chomp» puede traducirse como «masticar».

final, debe saludar a la persona usando su nombre completo.

- Escribe un programa que pregunta por el número favorito de una persona. Haz que tu programa le sume 1 al número y entonces sugiera el resultado como un *más grande y mejor* número favorito. (Pero hazlo con tacto.)

Una vez que hayas terminado esos dos programas (y cualquier otro que quisieras intentar), aprendamos algo más (y un poco más) acerca de los métodos.

5. Más acerca de los métodos

Hasta ahora hemos visto un pequeño número de diferentes métodos, **`puts`**, **`gets`**, y así por el estilo (***Examen rápido**: imenciona todos los métodos que hemos visto hasta ahora! Son diez de ellos; la respuesta está más abajo.*), pero no hemos hablado acerca de lo que son los métodos.

Aunque realmente, eso *es* lo que son: cosas que hacen algo. Si los objetos (como las cadenas, los enteros y los flotantes) son los sustantivos en el lenguaje Ruby, entonces los métodos son como los verbos. Y, justo como en el español, no puedes tener un verbo sin un sustantivo que *realice* el verbo. Por ejemplo, sonar no es sólo algo que pase; un reloj de pared (o de pulsera o algo) tiene que hacerlo. En español diríamos, «El reloj suena». En Ruby diríamos **`reloj.suena`** (asumiendo que **`reloj`** es un objeto en Ruby, claro). Los programadores podrían decir que estamos «llamando el método **`suena`** de **`reloj`**» o que «llamamos **`suena`** sobre **`reloj`**».

¿Hiciste el examen rápido? ¡Bien! Bueno, estoy seguro recordaste los métodos **`puts`**, **`gets`** y **`chomp`**, ya que los acabamos de ver. También es probable que hayas recordado nuestros métodos de conversión: **`to_i`**, **`to_f`** y **`to_s`**. Sin embargo, ¿recordaste los últimos cuatro? ¡No son otros más que nuestros viejos compañeros aritméticos **`+`**, **`-`**, **`*`** y **`/`**!

Y, como decía, justo como cada verbo necesita un sustantivo, así cada método necesita un objeto. Usualmente es fácil decir qué objeto está realizando un método: es lo que viene justo antes del punto, como en el ejemplo de **`reloj.suena`**, o en **`101.to_s`**. Algunas veces, sin embargo, no es tan obvio; como con los métodos aritméticos. Resulta que **`5 + 5`** es solo un atajo para escribir **`5.+(5)`**. Por ejemplo:

```
puts 'hola '.+('mundo')
puts (10.*(9)).+(9)
```

```
hola mundo
99
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

No es muy bonito, así que no lo volveremos a escribir de esa forma; sin embargo, es importante comprender lo que *realmente* está pasando. (En mi máquina, eso también me da una *advertencia*: **`warning: parenthesize argument(s) for future`**

`version`[11]. Aun así, el código se ejecutó bien, pero me dice que está teniendo problemas interpretando lo que quiero decir, y que use más paréntesis en un futuro). Esto también nos da un mejor entendimiento de porqué podemos hacer `'cerdo'*5` pero no podemos hacer `5*'cerdo'`: `'cerdo'*5` le dice a `'cerdo'` que haga la multiplicación, en cambio `'cerdo'*5` le dice a `5` que haga la multiplicación. `'cerdo'` sabe cómo hacer `5` copias de sí mismo y cómo ponerlas todas juntas; sin embargo, `5` la tendrá mucho más difícil para hacer `'cerdo'` copias de *sí mismo* y sumarlas todas.

Y, por supuesto, aún tenemos a **`puts`** y a **`gets`** por explicar. ¿Dónde están sus objetos? En español, algunas veces puedes dejar fuera el sujeto; por ejemplo, si un villano grita «¡Muere!», el sujeto implícito es a quien le está gritando. En Ruby, si yo digo `puts 'ser o no ser'`, lo que en realidad estoy diciendo es `self.puts 'ser o no ser'`. Entonces, ¿qué es `self`? Es una variable especial que apunta a cualquier objeto dentro del que te encuentras. Aún no sabemos cómo estar *dentro* de un objeto pero, hasta que lo averigüemos, siempre vamos a estar dentro de un gran objeto que es... ¡el programa completo! Y afortunadamente para nosotros, el programa tiene algunos métodos propios, como **`puts`** y **`gets`**. Observa esto:

```
noPuedoCreerQueCreeUnaVariableTanLargaParaApuntarA3 = 3
puts noPuedoCreerQueCreeUnaVariableTanLargaParaApuntarA3
self.puts noPuedoCreerQueCreeUnaVariableTanLargaParaApuntarA3
```

```
3
3
```

Si no seguiste todo lo anterior, está bien. Lo importante de todo eso es que cada método está siendo hecho por algún objeto, aun si no tiene un punto frente a él. Si entiendes eso, está todo bien.

Métodos creativos para texto

Aprendamos algunos métodos divertidos para cadenas de texto. No tienes que memorizarlos todos; puedes volver a revisar ésta página si los llegas a olvidar. Sólo quiero mostrarte una *pequeña* parte de lo que se puede hacer con el texto. De hecho, yo mismo no puedo recordar ni la mitad de los métodos de cadenas de texto — pero eso está bien, porque hay buenas referencias en la Internet con todos los métodos para cadenas listados y explicados. (Te enseñaré dónde encontrarlos al final de éste tutorial).

[11] «Advertencia: coloca los argumentos dentro de paréntesis para futuras versiones [del programa]»

De verdad, ni siquiera *quiero* conocer todos los métodos para cadenas; es como querer conocer todas las palabras de un diccionario. Puedo hablar español bien sin conocer cada palabra del diccionario... y ¿no es ese el punto de que exista un diccionario? ¿Qué no *tengas* que conocer lo que hay en él?

Así que, nuestro primer método para texto es **`reverse`**, que da una versión de atrás hacia adelante de una cadena.

```
var1 = 'alto'
var2 = 'estresado'
var3 = '¿Puedes decir ésta oración al revés?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

```
otla
odasertse
?séver la nóicaro atsé riced sedeuP¿
alto
estresado
¿Puedes decir ésta oración al revés?
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Como puedes ver, **`reverse`** no voltea a la cadena original; sólo hace una versión de ella de atrás hacia adelante. Es por es que **`var1`** sigue siendo **`'alto'`** aún después de que llamó a **`reverse`**.

Otro de los métodos de las cadenas es **`length`**, que nos dice el número de caracteres (incluyendo espacios) en la cadena.

```
puts '¿Cuál es tu nombre completo?'
nombre = gets.chomp
puts '¿Sabías que hay ' + nombre.length + 'caracteres en tu nombre, '
+ nombre + '?'
```

```
¿Cuál es tu nombre completo?
Christopher David Pine
#<TypeError: can't convert Fixnum into String>[12]
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Oh-oh! Algo salió mal y parece que pasó en algún lugar después de la línea **`nombre = gets.chomp`**... ¿Ves el problema? Prueba si puedes averiguarlo.

El problema es con **`length`**: ya que devuelve un número, pero lo que queremos es texto. Es muy fácil, sólo agregaremos un **`to_s`** (y cruzaremos los dedos):

```
puts '¿Cuál es tu nombre completo?'
nombre = gets.chomp
puts '¿Sabías que hay ' + nombre.length.to_s + ' caracteres en tu nombre, '
+ nombre + '?'
```

```
¿Cuál es tu nombre completo?
Christopher David Pine
¿Sabías que hay 22 caracteres en tu nombre, Christopher David Pine?
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

No, no lo sabía. **Nota**: ese es el número de *caracteres* en mi nombre, no el número de *letras* (puedes contarlas). Creo que podríamos escribir un programa que pregunte por cada una de las partes de tu nombre individualmente y entonces sume las longitudes... ¡hey!, ¿por qué no lo haces? Vamos, yo esperare.

¿Lo hiciste? ¡Bien! Es un bonito programa, ¿no es así? Después de unos cuantos capítulos más, te asombraras de lo que puedes hacer.

También hay algunos métodos de cadenas que pueden cambiar las formas del texto (mayúsculas, minúsculas) en tus cadenas. **`upcase`** cambia todas las letras minúsculas a mayúsculas, y **`downcase`** cambia cada mayúscula a minúscula. **`swapcase`** cambia la forma de cada letra en la cadena (sí es mayúscula la vuelve minúscula y viceversa), y finalmente, **`capitalize`** es como **`downcase`**, excepto que cambia el primer carácter a mayúscula (si es una letra).

[12] Literalmente se traduciría como: «ErrorDeTipo: no puedo convertir NúmeroFijo en Cadenas».

```
letras = 'aAbBcCdDeE'

puts letras.upcase
puts letras.downcase
puts letras.swapcase
puts letras.capitalize
puts ' a'.capitalize
puts letras
```

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
 a
aAbBcCdDeE
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Cosas bastante usuales. Como puedes ver en la línea **`puts ' a'.capitalize`**, el método **`capitalize`** sólo cambia a mayúscula el primer *carácter*, no la primera *letra*. También, como hemos visto antes, a través de todas esas llamadas a métodos, las letras originales permanecen sin cambio. No deseo elaborar mucho ese punto, pero es importante entenderlo. Hay algunos métodos *sí* cambian el objeto asociado, pero no hemos visto alguno hasta ahora, y no lo haremos durante un tiempo.

Los últimos métodos creativos para texto que veremos son los de formato visual. El primero, **`center`**, añade espacios al principio y final de la cadena para centrarla. Sin embargo, justo como debes decirle a **`puts`** que es lo que quieres imprimir, y a **`+`** lo que quieres añadir, debes decirle a **`center`** que tan ancha deseas que sea la cadena a centrar. Así, si yo quisiera centrar las líneas de un poema, lo haría de la siguiente forma:

```
anchoDeLinea = 50
puts('Como todas las cosas están llenas de mi alma'.center(anchoDeLinea))
puts(   'emerges de las cosas, llena del alma mía.'.center(anchoDeLinea))
puts(    'Mariposa de sueño, te pareces a mi alma,'.center(anchoDeLinea))
puts(       'y te pareces a la palabra melancolía.'.center(anchoDeLinea))
```

```
            Como todas las cosas están llenas de mi alma
```

```
emerges de las cosas, llena del alma mía.
Mariposa de sueño, te pareces a mi alma,
y te pareces a la palabra melancolía.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Hmmm... no estoy seguro si así va ese fragmento de ese poema, pero soy muy flojo para buscarlo. (Además, quería alinear la parte de **`.center(anchoDeLínea)`**, por lo que puse esos espacios extra antes de las cadenas. Esto es porque pienso que se ve mejor de esa forma.

Los programadores usualmente tienen fuertes sentimientos acerca de como les gusta escribir el código de los programas y frecuentemente está en desacuerdo. Entre más programas, más desarrollas tu propio estilo.) Hablando de ser flojo, la flojera no es siempre algo malo en la programación. Por ejemplo, ¿ves cómo guardé la longitud de la línea para alinear el poema en la variable **`anchoDeLínea`**? Esto es por si más tarde quiero que el poema se centre en una línea más amplia, sólo tengo que cambiar la primera línea de código del programa, en lugar de cada línea encargada de hacer el centrado. Con un poema muy largo, esto podría ahorrarme mucho tiempo. Éste tipo de flojera es realmente una virtud en cuanto a programación.

Así que, acerca de ese centrado... puedes haber notado que no es tan bonito como un procesador de palabras lo hubiera hecho. Si realmente deseas un centrado perfecto (y tal vez una fuente de texto más bonita), ¡entonces deberías usar un procesador de palabras! Ruby es una maravillosa herramienta, pero ninguna herramienta es la herramienta adecuada para *todos* los trabajos.

Los otros dos métodos para el formato de cadenas de texto son **`ljust`** y **`rjust`**, que significan *justificar a la izquierda* y *justificar a la derecha*. Son similares a **`center`**, excepto que colocan espacio extra a la derecha y a la izquierda, respectivamente. Veamos a los tres en acción:

```
anchoDeLinea = 44
cadena = '--> texto <--'

puts cadena.ljust(anchoDeLinea)
puts cadena.center(anchoDeLinea)
puts cadena.rjust(anchoDeLinea)
puts cadena.ljust(anchoDeLínea / 2) + cadena.rjust(anchoDeLínea / 2)
```

```
--> texto <--
                --> texto <--
                                --> texto <--
--> texto <--                   --> texto <--
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Algunas cosas para intentar

- Escribe el programa «Jefe enojado». Debe preguntar de forma ruda que es lo que quieres. Lo que sea que le pidas, el jefe enojado debe contestarte gritando y después, despedirte. Así que, por ejemplo, si tú escribes `Quiero un aumento`. él debe contestar gritando: `¿¡¿QUÉ QUIERES DECIR CON «QUIERO UN AUMENTO»?!? ¡¡ESTÁS DESPEDIDO!!`

- Aquí hay algo para que puedas jugar un poco más con `center`, `ljust` y `rjust`. Escribe un programa que desplegará una «Tabla de contenidos» de forma que se vea como ésta:

```
            Tabla de contenidos

Capítulo 1: Números                     página 1
Capítulo 2: letras                      página 72
Capítulo 3: Variables                   página 118
```

6. Matemáticas avanzadas

(Éste capítulo es totalmente opcional. Asume un grado moderado de conocimiento matemático. Si no estás interesado, puedes continuar directamente al capítulo sobre control de flujo sin ningún problema. Sin embargo, un repaso rápido sobre la sección de ***números aleatorios*** *podría ser de utilidad.)*

No hay tantos métodos para números como los hay para cadenas de caracteres (aun así, no los tengo todos en la memoria). Aquí veremos al resto de los métodos aritméticos, un generador de números aleatorios y el objeto **`Math`** con sus métodos trigonométricos y trascendentales.

Más aritmética

Los otros dos métodos aritméticos son **`**`** (potenciación) y **`%`** (módulo). Así que si quieres escribir «cinco al cuadrado» en Ruby, lo escribirías como `5 ** 2`. También puedes usar números reales como exponente, si quieres la raíz cuadrada de 5, puedes escribirlo como `5 ** 0.5`. El método modulo te da el remanente de una división. Así que, por ejemplo, si divido 7 entre 3, obtengo 2 con un remanente de 1.

Veamos cómo funcionan en un programa:

```
puts 5 ** 2
puts 5 ** 0.5
puts 7 / 3
puts 7 % 3
puts 365 % 7
```

```
25
2.23606797749979
2
1
1
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

De la última línea aprendemos que un año (no bisiesto) tiene cierto número de semanas más un día. Así que si tu cumpleaños es (o fue) el martes éste año, será en miércoles el siguiente. También puedes usar números reales con el método módulo.

Básicamente funciona de la única forma sensible que podría... pero te dejaré jugar con eso.

Hay un último método que mencionar antes de ver la generación de números aleatorios: **`abs`**. Sólo devuelve el valor absoluto de un número:

```
puts (5 - 2).abs
puts (2 - 5).abs
```

```
3
3
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Números aleatorios

Ruby incluye un bonito generador de números aleatorios. El método para obtener un número elegido al azar es **`rand`**. Si llamas a **`rand`** simplemente así, obtendrás un número fraccionario igual o mayor que `0` y menor que `1.0`. Si le das a **`rand`** un entero (`5`, por ejemplo), te devolverá un entero mayor o igual que `0` y menor que `5` (cinco posibles números del `0` al `4`).

Vemos a **`rand`** en acción.

```
puts rand
puts rand
puts rand
puts rand(100)
puts rand(100)
puts rand(100)
puts rand(1)
puts rand(1)
puts rand(1)
puts rand(99999999999999999999999999999999999999999999999999999999)
puts 'El hombre del clima dijo que hay '+rand(101).to_s+'% de probabilidad'
puts 'de lluvia, pero nunca debes de confiar en el hombre del clima.'
```

```
0.866769322351658
0.155609260113273
```

```
0.208355946789083
61
46
92
0
0
0
229824775081318602319541087738875238616006939895184956998 62
El hombre del clima dijo que hay 47% d probabilidad
de lluvia, pero nunca debes de confiar en el hombre del clima.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Nota que he usado **`rand(101)`** para obtener números entre 0 y 100, y que **`rand(1)`** siempre regresa 0. No comprender el rango de posibles valores devueltos es el error más grande que he visto a otras personas realizar con **`rand`**; aun programadores profesionales; aun en productos terminados que puedes comprar en una tienda. Una vez tuve un reproductor de CDs que al ser colocado en modo de «reproducción aleatoria», tocaba todas las pistas excepto la última... (Me pregunto, ¿qué hubiera pasado si hubiera colocado con un CD con sólo una pista en él?)

Algunas veces podrías querer que **`rand`** regrese los *mismos* números aleatorios en la misma secuencia en dos diferentes ejecuciones de tu programa. (Por ejemplo, una vez utilice un generador de números aleatorios para crear un mundo generado automáticamente para un juego de computadora. Si encontraba un mundo que realmente me gustara, tal vez me gustaría jugar en el de nuevo, o enviarlo a un amigo.) Para lograr esto, necesitas configurar un valor *semilla*, lo cual puedes hacer con **`srand`**. Algo así:

```
srand 1776
puts rand(100)
puts rand(100)
puts rand(100)
puts rand(100)
puts rand(100)
puts ''
srand 1776
puts rand(100)
puts rand(100)
```

```
puts rand(100)
puts rand(100)
puts rand(100)
```

```
24
35
36
58
70

24
35
36
58
70
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Hará lo mismo cada vez que utilices el mismo número como semilla. Si quieres volver a obtener diferentes números otra vez (como si nunca hubieras usado **`srand`**), entonces llama **`srand 0`**. Esto le da como semilla un número realmente raro, usando (entre otras cosas) la hora actual de tu computadora, con precisión de milisegundos.

El objeto `Math`

Por último, veamos el objeto **`Math`**. También podríamos saltar directamente dentro de él:

```
puts Math::PI
puts Math::E
puts Math.cos(Math::PI / 3)
puts Math.tan(Math::PI / 4)
puts Math.log(Math::E ** 2)
puts (1 + Math.sqrt(5)) / 2
```

```
3.14159265358979
2.71828182845905
```

```
0.5
1.0
2.0
1.61803398874989
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Lo primero que tal vez notaste fue la notación `::`. Explicar el *operador de alcance* (que es lo eso es) está realmente más allá del, eh... del alcance de éste tutorial. No es broma. Lo juro. Es suficiente decir que puedes usar `Math::Pi` justo como lo esperarías.

Como puedes ver, `Math` tiene todas las cosas que podrías esperar de una calculadora científica decente. Y como siempre, los números fraccionarios están *realmente cerca* de ser las respuestas correctas.

Así que ahora, ¡fluyamos!

7. Control del flujo

Ahhh, el control de flujo. Aquí es donde todo se condensa. Aun cuando éste capítulo es más corto y sencillo que el capítulo sobre métodos, abrirá un completo mundo de posibilidades de programación. Después de éste capítulo, seremos capaces de escribir programar realmente interactivos; en el pasado hemos hecho programas que *dicen* diferentes cosas dependiendo de lo que escribías en el teclado, pero después de éste capítulo realmente *harán* diferentes cosas también. Pero antes de que podamos hacer eso, necesitamos ser capaces de comparar los objetos en nuestros programas. Necesitamos...

Métodos de comparación

Vamos a darnos prisa por ésta parte para poder llegar a la siguiente sección, **ramificación**[13], donde ocurren todas las cosas interesantes. Así que, para ver si un objeto es mayor o menor que otro, usamos los métodos **>** y **<**, de ésta forma:

```
puts 1 > 2
puts 1 < 2
```

```
false
true
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Sin problemas. De igual forma podemos encontrar si un objeto es mayor-o-igual (o menor-o-igual) que otro con los métodos **>=** y **<=**:

```
puts 5 >= 5
puts 5 <= 4
```

```
true
false
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

[13] Del término «branching» en inglés.

Y, finalmente, podemos ver si dos objetos son iguales o no usando **==** (que significa «¿son iguales?») y **!=** (que significa «¿son diferentes?»). Es importante no confundir **=** con **==**. **=** es para decir a una variable que apunte a un objeto (asignación), y **==** es para hacer la pregunta «¿Son estos dos objetos iguales?»:

```
puts 1 == 1
puts 2 != 1
```

```
true
true
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Por supuesto, también podemos comparar cadenas de caracteres. Cuando las cadenas son comparadas, se hace utilizando su *orden lexicográfico*, lo que básicamente significa su orden de aparición en un diccionario, **gato** aparece antes que **perro**, así que:

```
puts 'gato' < 'perro'
```

```
true
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Sin embargo, hay un detalle: la forma en la que las computadoras usualmente hacen las cosas. Ellas usualmente ordenan las letras mayúsculas antes de las letras minúsculas. (Así es como almacenan las letras en las fuentes, por ejemplo: primero todas las letras en mayúsculas, después todas las letras en minúsculas.) Esto significa que la computadora pensará que **'Zoológico'** viene antes que **'hormiga'**, así que si quieres averiguar que palabra viene primero en un diccionario real, está seguro de usar **downcase** (o **upcase** o **capitalize**) en ambas palabras antes de intentar compararlas.

Una última cosa antes de ver el tema de **ramificación**: Los métodos de comparación no nos están dando las cadenas de texto **'true'** y **'false'**; nos están dando los objetos especiales **true** y **false**. (Por supuesto, **true.to_s** nos devuelve **'true'**, que es como **puts** imprimió **'true'**.) **true** y **false** se usan todo el tiempo en...

Ramificación

La ramificación (o bifurcación) es un concepto simple, pero poderoso. De hecho, es tan simple que apuesto a que no tendré que explicarlo para nada; sólo te lo mostraré:

```
puts 'Hola, ¿cómo te llamas?'
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'
if nombre == 'Chris'
  puts '¡Qué hermoso nombre!'
end
```

```
Hola, ¿cómo te llamas?
Chris
Hola, Chris.
¡Qué hermoso nombre!
```

Pero si le damos un nombre diferente...

```
Hola, ¿cómo te llamas?
Chewbacca
Hola, Chewbacca[14].
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Y eso es la ramificación. Si lo que viene después de **`if`** da como resultado **`true`**, ejecutamos el código entre **`if`** y **`end`**. Si lo que viene después **`if`** da como resultado **`false`**, no lo hacemos. Así de simple.

Identé el código entre **`if`** y **`end`** sólo porque creo que es más fácil que darse cuenta de la ramificación de esa forma. Casi todos los programadores lo hacen así, sin importar el lenguaje de programación que estén utilizando. Puede no parecer de mucha ayuda en éste ejemplo simple, pero conforme las cosas se vuelven más complejas, la identación hace una gran diferencia.

Algunas veces, quisiéramos que un programa hiciera una cosa si una expresión es **`true`**, e hiciera otra si la misma expresión es **`false`**. Para eso está **`else`**:

[14] Personaje ficticio del universo de «La Guerra de las Galaxias».

```
puts 'Soy un adivinador de fortunas. Dime tu nombre:'
nombre = gets.chomp
if nombre == 'Chris'
  puts 'Veo grandes cosas en tu futuro.'
else
  puts 'Tu futuro es... ¡Oh, mira la hora!'
  puts 'Realmente me tengo que ir, ¡lo siento!'
end
```

```
Soy un adivinador. Dime tu nombre:
Chris
Veo grandes cosas en tu futuro.
```

Ahora intentemos con un nombre diferente...[15]

```
Soy un adivinador. Dime tu nombre:
Ringo
Tu futuro es... ¡Oh, mira la hora!
Realmente me tengo que ir, ¡lo siento!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

La ramificación es como llegar a un cruce en el camino de nuestro código: ¿Tomamos el camino para la persona cuyo **`nombre == 'Chris'`** o, de otra forma, tomamos el otro camino?

Y justo como en las ramas de un árbol, puedes tener ramas que tienen ramas dentro de sí mismas:

```
puts 'Hola y bienvenido a la clase de Inglés del sexto grado.'
puts 'Yo soy la señorita Dulcinea. ¿Y tu nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  puts 'Por favor toma asiento, ' + nombre + '.'
else
  puts '¿' + nombre + '? Quieres decir ' + nombre.capitalize + ', ¿no es así?'
  puts '¿No sabes ni siquiera cómo escribir tu propio nombre?'
```

[15] Ringo Starr, músico y actor, ex-baterista del famoso grupo inglés The Beatles.

```
  respuesta = gets.chomp

  if respuesta.downcase == 'sí'
       puts '¡Jum! Bueno, ¡siéntate!'
  else
       puts '¡FUERA DE AQUÍ!'
  end
end
```

```
Hola y bienvenido a la clase de Inglés del sexto grado.
Yo soy la señorita Dulcinea. ¿Y tu nombre es...?
chris
¿chris? Quieres decir Chris, ¿no es así?
¿No sabes ni siquiera cómo escribir tu propio nombre?
sí
¡Jum! Bueno, ¡siéntate!
```

Bien, lo pondré en mayúsculas...

```
Hola y bienvenido a la clase de Inglés del sexto grado.
Yo soy la señorita Dulcinea. ¿Y tu nombre es...?
Chris
Por favor, toma asiento Chris.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Algunas veces puede ser confuso intentar comprender a donde van todos los **`if`**, **`else`** y **`end`**. Lo que yo hago es escribir el **`end`** *al mismo tiempo* que escribo su correspondiente **`if`**, por lo que es así como lucia el programa cuando comencé a escribirlo:

```
puts 'Hola y bienvenido a la clase de Inglés del sexto grado.'
puts 'Yo soy la señorita Dulcinea. ¿Y tu nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
else
end
```

Entonces lo llene con *comentarios*, cosas en el código que la computadora ignorará:

```
puts 'Hola y bienvenido a la clase de Inglés del sexto grado.'
puts 'Yo soy la señorita Dulcinea. ¿Y tu nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  # Ella es amable.
else
  # Ella se molesta.
end
```

Todo lo que viene después de un **#** es considerado un comentario (claro, a menos que esté dentro de una cadena de caracteres). Después de eso, remplacé los comentarios con código que funciona.

A algunas personas les gusta dejar los comentarios; personalmente, creo que código bien escrito usualmente habla por sí mismo. Solía utilizar más comentarios, pero entre más «fluido» me volvía en Ruby, menos los usaba. Hoy en día los encuentro como distracciones la mayor parte del tiempo. Es mi elección personal, tú encontrarás tu (usualmente siempre en evolución) estilo personal. Así que mi siguiente paso parecía algo así:

```
puts 'Hola y bienvenido a la clase de Inglés del sexto grado.'
puts 'Yo soy la señorita Dulcinea. ¿Y tu nombre es...?'
nombre = gets.chomp

if nombre == nombre.capitalize
  puts 'Por favor toma asiento, ' + nombre + '.'
else
  puts '¿' + nombre + '?  Quieres decir ' + nombre.capitalize + ', ¿no es así?'
  puts '¿No sabes ni siquiera cómo escribir tu propio nombre?'
  respuesta = gets.chomp

  if respuesta.downcase == 'sí'
  else
  end
end
```

De nuevo, escribí el **if**, el **else** y el **end** el mismo tiempo. Realmente me ayuda a

recordar «donde me encuentro dentro» del código. También hace parecer el trabajo más fácil porque puedo concentrarme en una pequeña parte, como en llenar el código entre el `if` y después dentro del `else`.

El otro beneficio de hacerlo de ésta manera es que la computadora puede comprender el programa en cualquier estado. Cada una de las versiones sin terminar del programa que he mostrado podría ser ejecutada. No estaban terminados, pero eran programas funcionales. De ésta forma podría probarlo conforme lo iba escribiendo, lo que me ayudo a ver cómo iba quedando y que era lo que aún necesitaba trabajo. ¡Cuándo pasaba todas las pruebas es como sabía que había terminado!

Estos consejos te ayudarán a escribir programas con ramificaciones, pero también ayudarán con otro tipo principal de flujo de control:

Ciclos

Usualmente querrás que tu computadora haga la misma cosa una y otra vez, después de todo para eso se supone que son buenas las computadoras.

Cuando le dices a tu computadora que se mantenga repitiendo algo, también debes decirle el momento en que debe detenerse. Las computadoras nunca se aburren, por lo que si no les indicas que se detengan, no lo harán. Nos aseguramos que esto no ocurra indicando a la computadora que repita ciertas partes del código *mientras* cierta condición es verdadera. Esto funciona de manera muy similar a como funciona `if`:

```
comando = ""

while comando != "adiós"
  puts comando
  comando = gets.chomp
end

puts "¡Vuelve pronto!"
```

```
¿Hola?
¿Hola?
¡Hola!
¡Hola!
```

```
Mucho gusto en conocerte.
Mucho gusto en conocerte.
Oh... ¡qué amable!
Oh... ¡qué amable!
adiós
¡Vuelve pronto!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Y eso es un ciclo. (Seguramente habrás notado la línea en blanco al principio de la salida; es del primer **`puts`**, antes del primer **`gets`**. ¿Cómo podrías cambiar el programa para deshacerte de esa primera línea? ¡Pruébalo! ¿Funciona *exactamente* cómo el programa de arriba a excepción de la línea en blanco?)

Los ciclos (o bucles) te permiten hacer todo tipo de cosas interesantes, como estoy seguro puedes imaginar. Sin embargo, te pueden causar problemas si cometes algún error. ¿Qué pasa si tu computadora se queda atrapada en un ciclo infinito? Si creen que eso puede haber pasado, sólo mantén presionada la tecla **`Ctrl`** y presiona la tecla **`C`**.

Antes de que comencemos a jugar con los ciclos, sin embargo, vamos a aprender algunas pocas cosas que harán nuestro trabajo más fácil.

Un poco de lógica

Revisemos nuestro primer programa con ramificaciones de nuevo. ¿Qué pasaría si mi esposa viniera a casa, viera el programa, lo probara y no el programa no le dijera que su nombre es hermoso? No quisiera herir sus sentimientos (o dormir en el sofá), así que vamos a reescribirlo.

```
puts 'Hola, ¿cómo te llamas?'
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'
if nombre == 'Chris'
  puts '¡Qué hermoso nombre!'
else
  if nombre == 'Katy'
     puts '¡Qué hermoso nombre!'
  end
end
```

```
Hola, ¿cómo te llamas?
Katy
Hola, Katy.
¡Qué hermoso nombre!
```

Bueno, funciona. Pero no es un programa muy bonito. ¿Por qué no? Bueno, la mejor regla que jamás aprendí en programación fue la regla *DRY*[16]: «No te repitas a ti mismo». Probablemente podría escribir un pequeño libro de porque es una regla tan buena. En nuestro caso, repetimos la línea **`puts '¡Qué hermoso nombre!'`**.

¿Por qué es tan importante? Bueno, ¿qué hubiera pasado si hubiera cometido un error al reescribirlo? ¿Qué pasaría si quisiera cambiar la frase de **`'hermoso'`** a **`'adorable'`** en ambas líneas? ¿Soy flojo, recuerdan? Básicamente, si quiero que el programa haga la misma cosa cuando obtenga **`'Chris'`** o **`'Katy'`**, entonces realmente debería hacer *la misma cosa*:

```
puts 'Hola, ¿cómo te llamas?'
nombre = gets.chomp
puts 'Hola, ' + nombre + '.'
if (nombre == 'Chris' or nombre == 'Katy')
  puts '¡Qué hermoso nombre!'
end
```

```
Hola, ¿cómo te llamas?
Katy
Hola, Katy.
¡Qué hermoso nombre!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Mucho mejor. Para lograr hacerlo funcionar, utilice **`or`**. Los otros *operadores lógicos* son **`and`** y **`not`**. Siempre es buena idea usar paréntesis cuando trabajemos con estos. Veamos cómo funcionan:

```
yoSoyChris      = true
```

[16] Acrónimo del inglés «Don't Repeat Yourself».

```
yoSoyMorado     = false
meGustaLaComida = true
yoComoRocas     = false

puts (yoSoyChris and meGustaLaComida)
puts (meGustaLaComida and yoComoRocas)
puts (yoSoyMorado and meGustaLaComida)
puts (yoSoyMorado and yoComoRocas)
puts
puts (yoSoyChris or meGustalaComida)
puts (meGustaLaComida or yoComoRocas)
puts (yoSoyMorado or meGustaLaComida)
puts (yoSoyMorado or yoComoRocas)
puts
puts (not yoSoyMorado)
puts (not yoSoyChris)
```

```
true
false
false
false

true
true
true
false

true
false
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

El único que probablemente podría engañarte es **`or`**. En español, frecuentemente usamos «o» para indicar «uno o el otro, pero no ambos». Por ejemplo, tu madre podría decir «Para el postre, puedes elegir pay o pastel». ¡Ella *no* quiere decir que puedes comer ambos! Una computadora, por otro lado, usa **`or`** para decir «uno o el otro, o ambos». (Otra forma de decir esto es, «al menos uno de estos es verdad».) Es por esto que las computadoras son más divertidas que las madres.

Algunas cosas para intentar

- «Un elefante se balanceaba...» Escribe un programa que imprima 99 estrofas de la letra de ésta clásica canción infantil: «Un elefante se balanceaba sobre la tela de una araña».[17]

- Escribe el programa de la abuela sorda. Para cualquier cosa que le digas a la abuela (esto es, cualquier cosa que escribas), ella debe responder con `¿¡QUÉ!?` `¡HABLA MÁS FUERTE HIJITO!`, a menos que se lo digas gritando (escribiendo todo en mayúsculas). Si gritas, ella podrá escucharte (o al menos eso creerá ella) y te responderá gritando `¡NO, NO DESDE 1938!` Para hacer el programa realmente creíble, haz que la abuela grite un año cualquiera al azar entre 1930 y 1950. (Ésta parte del programa es opcional y será mucho más fácil de realizar si lees la sección del generador de números aleatorios de Ruby al final del capítulo sobre métodos). No puedes dejar de hablar con la abuela hasta que grites `ADIÓS`.

  > ***Pista:*** *¡No te olvides de* `chomp`*! ¡*`'ADIÓS'` *con un Intro no es lo mismo que un* `'ADIÓS'` *sin uno!*
  >
  > ***Pista 2:*** *Trata de pensar acerca de que partes de tu programa tienen que repetirse una y otra vez. Todas ellas deben estar dentro de una iteración con* `while`.

- Extiende tu programa de la abuela sorda: ¿Qué hay si la abuela no quiere que te vayas? Cuando grites `ADIÓS`, ella puede pretender no oírte. Cambia tu programa anterior para que tengas que gritar `ADIÓS` tres veces *seguidas*. Asegúrate de probar tu programa: si dices `ADIÓS` tres veces pero no de forma consecutiva, debes de seguir hablando con la abuela.

- Años bisiestos: Escribe un programa que solicite un año inicial y un año final, y entonces imprima todos los años bisiestos entre esos dos años (incluyéndolos si también son años bisiestos). Los años bisiestos son divisibles entre cuatro (como 1984 y 2004). Sin embargo, los años divisibles entre 100 no son años bisiestos (como 1800 y 1900) ***a menos que*** también sean divisibles entre 400 (como 1600 y 2000, los que fueron de hecho, años bisiestos). *(Sí, es bastante confuso, pero no es tan confuso como tener el mes de Julio a mitad del invierno, que es lo que eventualmente pasaría).*

¡Cuando hayas terminado, toma un descanso! Ya has aprendido mucho.

[17] En Youtube podrás encontrar algunos vídeos con la canción.

¡Felicidades! ¿Estás sorprendido por el número de cosas que puedes decirle que haga a una computadora? Unos cuantos capítulos más y serás capaz de programar casi cualquier cosa. ¡De verdad! Sólo observa todas las cosas que puedes hacer ahora que no podrías hacer sin los ciclos y la ramificación.

Ahora, aprendamos algo acerca de un nuevo tipo de objetos, unos que guardan referencias a otros objetos: los arreglos.

8. Arreglos e iteradores

Escribamos un programa que nos pida introducir tantas palabras como deseemos (una palabra por línea, continuando hasta que presionemos **`Intro`** en una línea vacía), y que repita las palabras que escribimos en orden alfabético. ¿Está bien?

Así que... primero, buenooo... um... hmmm... Bueno, nosotros podríamos-er... um...

Sabes, no creo que podamos hacerlo. Necesitamos una forma de almacenar una cantidad desconocida de palabras y de mantener un registro de todas ellas juntas, de forma que no se mezclen con otras variables. Necesitamos colocarlas en algún tipo de lista. Necesitamos *arreglos*.

Un arreglo es sólo una lista en tu computadora. Cada casilla en la lista actúa como una variable: puedes ver hacia que objeto en particular apunta el casillero y puedes hacer que apunte a un objeto diferente. Veamos algunos arreglos:

```
[]
[5]
['Hello', 'Goodbye']

sabor = 'vainilla'            # Claro, éste no es un arreglo...
[89.9, sabor, [true, false]]  # ...pero éste sí lo es.
```

Primero tenemos un arreglo vacío, después un arreglo que contiene un sólo número, después un arreglo que contiene dos cadenas de texto. A continuación tenemos una asignación sencilla; y entonces un arreglo que contiene tres objetos, el último de los cuales es el arreglo **`[true, false]`**. Recuerda, las variables no son objetos, así que nuestro último arreglo realmente está apuntando a un número flotante, a una *cadena de caracteres*, y a un arreglo. Aún si hiciéramos que **`sabor`** apuntará a otra cosa, eso no cambiaría el arreglo.

Para ayudarnos a encontrar un objeto en particular dentro de un arreglo, a cada casilla se le asigna un número de índice. Los programadores (e incidentalmente, la mayoría de los matemáticos) comienzan a contar desde cero, por lo que la primera casilla del arreglo es la casilla cero. Aquí se muestra cómo hacemos referencia a los objetos en un arreglo:

```
nombres = ['Ada', 'Bella', 'Chris']

puts nombres
puts nombres[0]
puts nombres[1]
puts nombres[2]
puts nombres[3]  # Esto está fuera de rango.
```

```
Ada
Bella
Chris
Ada
Bella
Chris
nil
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Entonces, vemos que **`puts nombres`** imprime cada nombre en el arreglo **`nombres`**. Después, usamos **`puts nombres[0]`** para imprimir el primer nombre del arreglo, y **`puts nombres[1]`** para imprimir el segundo... Estoy seguro que esto parece confuso, pero *te* acostumbras. Sólo tienes que comenzar a *pensar* que el contar realmente comienza desde cero y dejar de usar palabras como «primero» y «segundo».

Si vas a disfrutar de una comida de 5 entradas, no te refieras a la «primera entrada»; habla sobre la «entrada cero» (y en tu mente piensa **`entrada[0]`**). Tienes 5 dedos en tu mano derecha, y sus números son 0, 1, 2, 3 y 4. Mi esposa y yo somos malabaristas. Cuando hacemos malabares con seis objetos, lo hacemos con los objetos del 0 al 5. Tenemos esperanzas de, en unos pocos meses, seremos capaces de agregar el objeto 6 (con lo que estaríamos haciendo malabares con 7 objetos entre nosotros).[18]

Por último, intentamos **`puts nombres[3]`**, sólo para ver lo que pasaría. ¿Estabas esperando un error? Algunas veces, cuando haces una pregunta, tu pregunta no tiene sentido (al menos para tu computadora); ahí es cuando obtienes un error. Otras veces, sin embargo, puedes hacer una pregunta y la respuesta es *nada*. ¿Qué hay en la casilla tres? Nada. ¿Cuál contenido de **`nombres[3]`**? **`nil`**: la forma en que Ruby dice «nada». **`nil`** es un objeto especial que básicamente significa «ningún otro objeto».

[18] En el tutorial original, el sr. Pine hacía referencia a usar el cardinal del número cero, pero como ese cardinal no existe en español, la oración fue removida.

Si toda ésta forma rara de numerar las casillas de los arreglos te pone nervioso, ¡no tengas miedo! Muy a menudo, podemos evitarlo completamente al utilizar varios métodos de arreglos, como éste:

El método `each`

`each` nos permite hacer algo (lo que sea que queramos) a *cada* objeto al que apunta un arreglo. Así que, si quisiéramos decir algo agradable sobre cada uno de los lenguajes dentro del arreglo de abajo, haríamos esto:

```
lenguajes = ['el inglés', 'el alemán', 'Ruby']

lenguajes.each do |lenguaje|
  puts '¡A mí me gusta ' + lenguaje +'!'
  puts '¿A ti no?'
end

puts '¡Ahora para Java!'
puts '...'
```

```
¡A mí me gusta el inglés!
¿A ti no?
¡A mí me gusta el alemán!
¿A ti no?
¡A mí me gusta Ruby!
¿A ti no?
'¡Ahora para Java!'
...
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¿Qué acaba de pasar? Bueno, fuimos capaces de ir a través de cada objeto en el arreglo sin usar ningún número, y eso es definitivamente agradable. Traduciendo al español, el programa de arriba se lee algo como: Para *cada* objeto en **`lenguajes`**, apunta la variable **`lenguaje`** al objeto y entonces haz todo lo que te digo hasta que llegues al final. (Como dato cultural, Java es otro lenguaje de programación. Es mucho más difícil de aprender que Ruby; usualmente un programa en Java será mucho más grande que un programa en Ruby que haga lo mismo).

Podrías estar pensando, «Esto es muy parecido a los ciclos sobre la que aprendimos antes». Sí, es similar. Una diferencia importante es que el método **`each`** es

sólo eso: un método. **while** y **end** (así como **do**, **if**, **else** y todas las otras palabras azules) no son métodos. Son parte fundamental Ruby como lenguaje, al igual que **=** y los paréntesis; en forma similar a los signos de puntuación en el español.

Pero **each** no es así, **each** es sólo otro método de los arreglos. Los métodos que al igual que **each**, «actúan como» ciclos, son llamados *iteradores*.

Una cosa a notar sobre los iteradores es que siempre van seguidos de **do** ... **end**. **while** e **if** nunca tienen un **do** cerca de ellos; sólo usamos **do** con iteradores.

Aquí tenemos otro pequeño y bonito iterador, pero no es un método de arreglos... ¡es un método de los números enteros!

```
3.times do
  puts "¡Hip-Hip-Hurra!"
end
```

```
¡Hip-Hip-Hurra!
¡Hip-Hip-Hurra!
¡Hip-Hip-Hurra!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Más métodos de arreglos

Hemos aprendido sobre **each**, pero aún hay muchos otros métodos de arreglos... ¡casi tantos como hay métodos de cadenas de texto! De hecho, algunos de ellos (como **length**, **reverse**, **+** y *****) funcionan justo como lo hacen para las cadenas, excepto que operan sobre las casillas de los arreglos en vez de las letras de una cadena. Otros, como **last** y **join**, son específicos de los arreglos. Aún otros, como **push** y **pop**, modifican los arreglos. Al igual que con los métodos de las cadenas, no tienes por qué recordar todos ellos, en tanto recuerdes donde informarte sobre ellos (justo en ésta sección).

Primero, vamos a ver **to_s** y **join**. **join** funciona muy parecido a como lo hace **to_s**, excepto que puede agregar texto entre los elementos de un arreglo. Observemos como lo hace:

```
alimentos = ['alcachofa', 'bollo', 'caramelo']

puts alimentos
puts
```

```
puts alimentos.to_s
puts
puts alimentos.join(', ')
puts
puts alimentos.join('  :)  ') + '  8)'

200.times do
   puts []
end
```

```
alcachofa
bollo
caramelo

alcachofabollocaralmelo

alcachofa, bollo, caramelo

alcachofa  :)  bollo  :)  caramelo 8)
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Como puedes ver, **`puts`** trata a los arreglos de forma diferente a otros objetos: simplemente llama a **`puts`** para cada uno de los elementos en el arreglo. Es por eso que intentar usar **`puts`** con un arreglo vacío `200` veces no hace nada; el arreglo apunta a nada, así que no hay algo que *poner*. (Hacer nada `200` veces aún es hacer nada.) Intenta usar **`puts`** con un arreglo que contenga otros arreglos; ¿se comporta como esperabas?

Por cierto, ¿te diste cuenta que no use la cadena vacía cuando quise imprimir una línea en blanco? Hace lo mismo.

Ahora observemos cómo funcionan **`push`**, **`pop`** y **`last`**. Los métodos **`push`** y **`pop`** son opuestos, como lo son **`+`** y **`-`**. **`push`** añade un objeto al final de tus arreglos, mientras que **`pop`** remueve el último objeto de tus arreglos (y te informa cual era). **`last`** es similar a **`pop`** en que te informa sobre que hay al final de un arreglo, excepto que no modifica el arreglo. De nuevo, **`push`** y **`pop`** *sí modifican los arreglos*:

```
favoritos = []
favoritos.push 'rocío sobre rosas'
favoritos.push 'whiskey sobre gatitos'
```

```
puts favoritos[0]
puts favoritos.last
puts favoritos.length
puts favoritos.pop
puts favoritos
puts favoritos.length
```

```
rocío sobre rosas
whiskey sobre gatitos[19]
2
whiskey sobre gatitos
rocío sobre rosas
1
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Algunas cosas para intentar

- Escribe el programa sobre el que hablamos justo al principio de éste capítulo. ***Pista**: Hay un adorable método de arreglos que devuelve la versión ordenada de un arreglo,* **`sort`**. *¡Úsalo!*
- Intenta escribir el programa anterior *sin* utilizar el método **`sort`**. Una parte importante de la programación es resolver problemas, ¡así que practica todo lo que puedas!
- Reescribe el programa «Tabla de contenidos» (del capítulo sobre métodos). Comienza el programa con un arreglo que contenga toda la información de la tabla de contenidos (nombres de capítulos, números de página, etc.). Después imprime la información con una bonita presentación.

Hasta ahora hemos aprendido un gran número de diferentes métodos. Es tiempo de aprender a hacer los nuestros.

[19] Parte de la letra de la popular canción "My Favorite Things", en la cual, a tono de broma se cambio la palabra "whishkers" (bigotes) por "whiskey".

9. Escribiendo tus propios métodos

Como hemos visto, los ciclos e iteradores nos permiten hacer lo mismo (ejecutar el mismo código) una y otra vez. Sin embargo, puede pasar que deseemos hacer algo cierto número de veces desde diferentes partes del programa.

Por ejemplo, digamos que escribíamos un programa de cuestionarios para un estudiante de psicología. De los estudiantes de psicología que he conocido y los cuestionarios que me han dado, seguramente sería algo como esto:

```
puts 'Hola y gracias por tomar un tipo de tiempo para ayudarme con éste'
puts 'experimento. Mi experimento trata de cómo se sienten las personas en'
puts 'relación con la comida mexicana. Sólo ten presente en tu mente la comida'
puts 'mexicana e intenta responder cada pregunta honestamente con un «sí» o un'
puts '«no». El experimento no tiene nada que ver con mojar la cama.'

# Hacemos éstas preguntas, pero ignoramos las respuestas.

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer tacos?'
  respuesta = gets.chomp.downcase

  if (respuesta == 'sí' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responde «sí» o «no».'
  end
end

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer burritos?'
  respuesta = gets.chomp.downcase

  if (respuesta == 'sí' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responde «sí» o «no».'
  end
end

# Pero sí ponemos atención a *ésta* respuesta.

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Aún mojas la cama?'
  respuesta = gets.chomp.downcase
```

```
  if (respuesta == 'sí' or respuesta == 'no')
    buenaRespuesta = true

    if respuesta == 'sí'
      mojaLaCama = true
    else
      mojaLaCama = false
    end
  else
    puts 'Por favor, responde «sí» o «no».'
  end
end

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer chimichangas?'
  respuesta = gets.chomp.downcase

  if (respuesta == 'sí' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responde «sí» o «no».'
  end
end

puts 'Sólo unas cuantas preguntas más...'

buenaRespuesta = false
while (not buenaRespuesta)
  puts '¿Te gusta comer sopapillas?'
  respuesta = gets.chomp.downcase

  if (respuesta == 'sí' or respuesta == 'no')
    buenaRespuesta = true
  else
    puts 'Por favor, responde «sí» o «no».'
  end
end

# Hace un montón de otras preguntas sobre comida mexicana.

puts
puts 'EXPLICACIÓN:'
puts 'Gracias por tomar tiempo para ayudarme con éste experimento. En realidad,'
puts 'el experimento no tiene nada que ver con comida mexicana. Es un'
puts 'experimento acerca de las personas que aún mojan la cama. La comida'
puts 'mexicana sólo estaba ahí como distracción, con la esperanza de que'
puts 'responderías con más honestidad. Gracias de nuevo.'
puts
puts mojaLaCama
```

```
Hola y gracias por tomar un tipo de tiempo para ayudarme con éste experimento. Mi
experimento trata de cómo se sienten las personas en relación con la comida
mexicana. Sólo ten presente en tu mente la comida mexicana e intenta responder cada
pregunta honestamente con un «sí» o un «no». El experimento no tiene nada que ver
con mojar la cama.

¿Te gusta comer tacos?
sí
¿Te gusta comer burritos?
sí
¿Aún mojas la cama?
¡claro qué no!
Por favor, responde «sí» o «no».
¿Aún mojas la cama?
NO
Sólo unas cuantas preguntas más...
¿Te gusta comer chimichangas?
sí
¿Te gusta comer sopapillas?
sí

EXPLICACIÓN:
Gracias por tomar tiempo para ayudarme con éste experimento. En realidad, el
experimento no tiene nada que ver con comida mexicana. Es un experimento acerca de
las personas que aún mojan la cama. La comida mexicana sólo estaba ahí como
distracción, con la esperanza de que responderías con más honestidad. Gracias de
nuevo.

false
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Ese fue un programa bastante largo, con muchas repeticiones. (Todas las secciones de código alrededor de preguntas sobre comida mexicana eran idénticas, y la pregunta sobre «mojar la cama» fue sólo ligeramente diferente. La repetición es mala. Aun así, no podemos usar un gran ciclo o un iterador, debido a que algunas veces tenemos cosas que queremos hacer entre preguntas. En situaciones como ésta, es mejor escribir un método.

Se hace así:

```
def muge
  puts 'muuuuuu...'
end
```

```

```

Uh... Nuestro programa no mugió. ¿Por qué no? Porque no le indicamos que lo hiciera. Le indicamos *cómo* mugir, pero nunca le dijimos que *lo* hiciera. Intentemos de nuevo:

```
def muge
  puts 'muuuuuu...'
end

muge
muge
puts 'coin-coin'
muge
muge
```

```
muuuuuu...
muuuuuu...
coin-coin
muuuuuu...
muuuuuu...
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Ahhh, mucho mejor! (Sólo en caso de que no hables francés, ese fue un pato francés a mitad del programa. En Francia, los patos dicen «*coin-coin*».)

Definimos el método **muge**. (Se usa que los nombres de métodos, como los nombres de variables, comiencen con una letra minúscula. Aunque, hay algunas excepciones como **+** o **==**.) Pero, ¿no tienen los métodos que estar siempre asociados con objetos? Bueno, sí, lo están, y en éste caso (al igual que **puts** y **gets**), el método está asociado con el objeto que representa el programa completo. En el siguiente capítulo veremos cómo agregar métodos a otros objetos. Pero primero...

Parámetros en los métodos

Tal vez hayas notado que algunos métodos (como **gets**, **to_s**, **reverse**...) pueden ser simplemente llamados sobre un objeto. Sin embargo, otros métodos (cómo **+**, **-**, **puts**...) toman *parámetros* para indicarle al objeto cómo realizar el método. Por ejemplo, no dirías simplemente **5+**, ¿verdad? Porque le estarías diciendo a **5** que sume, pero no le estarías diciendo *que* sumar.

Para añadir un parámetro a **muge** (digamos, el número de veces que debe mugir),

hacemos esto:

```
def muge numeroDeVeces
  puts 'muuuuuuu...' * numeroDeVeces
end

muge 3
puts 'oinc, oinc'
muge # Provocará un error porque falta un parámetro
```

```
muuuuuuu...muuuuuuu...muuuuuuu...
oinc, oinc
#<ArgumentError: wrong number of arguments (0 for 1)>[20]
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

numeroDeVeces es una variable que apunta al parámetro pasado al método. Lo diré de nuevo ya que es un poco confuso: **numeroDeVeces** es una variable que apunta al parámetro pasado al método. Así que, si escribo **muge** 3, el parámetro es 3 y la variable **numeroDeVeces** apunta a 3.

Como puedes ver, el parámetro es ahora *requerido*. Después de todo, ¿cómo puede **muge** multiplicar **'muuuuuuu...'** si no le das ningún parámetro? Tu pobre computadora no tiene idea.

Si los objetos en Ruby son como los sustantivos en español, y los métodos son como los verbos, entonces puedes pensar en los parámetros como adverbios (como en **muge**, donde el parámetro le dice *cómo* debe mugir) u otras veces como objetos directos (como con **puts**, donde el parámetro es *lo que* se imprime).

Variables locales

En el siguiente programa hay dos variables:

```
def duplicaEsto numero
  numeroPor2 = numero * 2
  puts numero.to_s + ' al doble es ' + numeroPor2.to_s
end

duplicaEsto 44
```

[20] Literalmente: ErrorDeArgumento: número equivocado de argumentos (0 de 1)

```
44 al doble es 88
```

Las variables son **`numero`** y **`numeroPor2`**. Ambas se encuentran dentro del método **`duplicaEsto`**. Éstas (y todas las variables que has visto hasta ahora) son *variables locales*. Esto significa que viven dentro del método y no pueden salir. Si intentas llamarlas afuera, obtendrás un error:

```
def duplicaEsto numero
  numeroPor2 = numero * 2
  puts numero.to_s + ' al doble es ' + numeroPor2.to_s
end

duplicaEsto 44
puts numeroPor2.to_s
```

```
44 al doble es 88
#<NameError: undefined local variable or method `numeroPor2' for #<StringIO:0x82ba21c>>[21]
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Variable local indefinida... De hecho, *sí* definimos esa variable local, pero no es local desde donde intentamos usarla; es local al método.

Esto podría parecer un inconveniente, pero en realidad es bastante bueno. Mientras que eso significa que no puedes acceder a variables dentro de los métodos, también significa que ellos no tienen acceso a *tus* variables, y por lo tanto, no las pueden afectar:

```
def pestecilla var
  var = nil
  puts 'Pestecilla: ¡Ja, ja! ¡He arruinado tu variable!'
end

var = 'Programa: ¡No puedes ni tocar mi variable!'
pestecilla var
puts var
```

```
Pestecilla: ¡Ja, ja! ¡He arruinado tu variable!
Programa: ¡No puedes ni tocar mi variable!
```

[21] Literalmente: ErrorDeNombre: variable local o método «numeroPor2» no definido.

Prueba el código en línea (da click en ➤ para ejecutar el programa)

En realidad hay *dos* variables llamadas **var** en ese pequeño programa: una dentro de la **pestecilla**, y la otra fuera de ella. Cuando llamamos a **pestecilla var**, en realidad sólo hicimos que ambas variables apuntaran a la misma cadena. Después, la **pestecilla** apunto su propia **var** *local* hacia **nil**, pero no hizo nada a la **var** fuera del método.

Retorno de valores

Probablemente te has dado cuenta que algunos métodos regresan algo cuando los llamas. Por ejemplo, **gets** *retorna* una cadena de caracteres (la cadena que escribiste con el teclado), y el método **+** en `5 + 3` (que es en realidad `5.+(3)`), retorna `8`. Los métodos aritméticos para números retornan números, y los métodos aritméticos para cadenas retornan cadenas.

Es importante entender la diferencia entre métodos retornando un valor a la parte del programa donde fueron llamados y la presentación de información en pantalla, como lo hace **puts**. Así, `5 + 3` retorna `8`, **no** imprime `8` en pantalla.

Entonces, *¿qué* retorna **puts**? No nos habíamos preocupado por eso, veamos ahora:

```
valorRetornado = puts 'Esto retorno puts:'
puts valorRetornado
```

```
Esto es lo que retorno puts:
nil
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Así que el primer **puts** retornó **nil**. Aunque no lo probamos directamente, el segundo **puts** también lo hizo; **puts** siempre retorna **nil**. Todos los métodos devuelven algo, aún si sólo es **nil**.

Toma un pequeño descanso y escribe un programa con el que averigües que retorna el método **muge**.

¿Te sorprendió? Bueno, así es como funciona: el valor que retorna un método es simplemente la última línea del método. En el caso de **muge**, esto significa que retorna **puts 'muuuuuuu...' * numeroDeVeces**, lo cual es **nil** debido a que **puts**

siempre devuelve `nil`. Si quisiéramos que todos nuestros métodos siempre retornaran la cadena `'submarino amarillo'`, tendríamos que poner *eso* en ellos al final:

```
def muge numeroDeVeces
  puts 'muuuuuuu...' * numeroDeVeces
  'submarino amarillo'
end

x = muge 2
puts x
```

```
muuuuuuu...muuuuuuu...
submarino amarillo
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Entonces, intentemos ese experimento de psicología de nuevo, pero ésta vez escribiremos un método para hacer las preguntas por nosotros. Necesitará recibir la pregunta como parámetro, retornando como resultado `true` si es respondida con `sí`, mientras que retorna como resultado `false` si es respondida con `no`. (Aun cuando la mayor parte del tiempo ignoraremos la respuesta, es una buena idea que el método retorne el resultado. Así también podremos usar la pregunta sobre mojar la cama.) Además, haré más cortas la bienvenida y la explicación, para que sea más fácil de leer:

```
def hacer_pregunta pregunta
  buenaRespuesta = false

  while (not buenaRespuesta)
    puts pregunta
    respuesta = gets.chomp.downcase

    if (respuesta == 'sí' or respuesta == 'no')
      buenaRespuesta = true

      if respuesta == 'sí'
        resultado = true
      else
         resultado = false
      end
    else
       puts 'Por favor, responde «sí» o «no».'
    end
```

```
  end

  resultado # lo que queremos retornar (true o false)
end

puts 'Hola y gracias por...'
puts

# ignoramos los valores retornados...
hacer_pregunta '¿Te gusta comer tacos?'
hacer_pregunta '¿Te gusta comer burritos?'
# pero guardamos el valor que es retornado ésta ocasión
mojaLaCama = hacer_pregunta '¿Aún mojas la cama?'
hacer_pregunta '¿Te gusta comer chimichangas?'
hacer_pregunta '¿Te gusta comer sopapillas?'
hacer_pregunta '¿Te gusta comer tamales?'
puts 'Sólo unas preguntas más...'
hacer_pregunta '¿Te gusta beber horchata?'
hacer_pregunta '¿Te gusta comer flautas?'

puts
puts 'EXPLICACIÓN:'
puts 'Te agradezco por...'
puts
puts mojaLaCama
```

```
Hola y gracias por...

¿Te gusta comer tacos?
sí
¿Te gusta comer burritos?
sí
¿Aún mojas la cama?
¡claro que no!
Por favor responde «sí» o «no».
¿Aún mojas la cama?
NO
¿Te gusta comer chimichangas?
sí
¿Te gusta comer sopapillas?
sí
¿Te gusta comer tamales?
sí
```

```
¿Te gusta beber horchata?
sí
¿Te gusta comer flautas?
sí

EXPLICACIÓN:
Te agradezco por...

false
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Nada mal, ¿eh? Fuimos capaces de agregar más preguntas (y agregar preguntas es *fácil* ahora), además de que nuestro programa es mucho más corto. ¡Es una gran mejora! – El sueño de un programador *flojo*.

Un ejemplo aún más grande

Creo que un método más a manera de ejemplo será de ayuda aquí. Llamaremos a éste **`numero_en_espanol`**. Tomará un número, como `22`, y retornará su versión en español (en éste caso, la cadena `'veintidós'`). Por ahora, hagamos que sólo funcione para números del `0` al `100`.

*(**NOTA**: Éste método usa un nuevo truco para retornar un valor de un método antes de llegar al final del mismo, usando la palabra `return`, e introduce también una variación a la ramificación: `elsif`. Debería ser claro cómo funcionan en el contexto.)*

```
def numero_a_espanol numero
  # Sólo queremos números del 0 al 100.
  if (numero < 0) or (numero > 100)
    return 'Por favor, proporciona un número entre 0 y 100.'
  end

  texto = '' # Ésta es la cadena que retornaremos.

  # Casos especiales: cuando numero es 0 o 100, devolvemos el texto apropiado.
  if numero == 0
    return 'cero'
  end
  if numero == 100
    return 'cien'
  end
```

```
# "restante" es lo que falta por escribir del número.
# "actual" es la parte que vamos a escribir ahora.
# "restante" y "actual" ¿Está claro? :)
restante = numero
actual   = restante / 10          # ¿Cuantas decenas se van a escribir?
restante = restante - actual * 10 # Sustraer esas decenas

if actual > 0
  if actual == 1 # Grupo del 10 al 19
    if restante < 7
      # Como no podemos escribir "diez y uno" en lugar de "once", tenemos que
      # establecer un comportamiento especial para éste caso y similares.
      if    restante == 0
        texto = texto + 'diez'
      elsif restante == 1
        texto = texto + 'once'
      elsif restante == 2
        texto = texto + 'doce'
      elsif restante == 3
        texto = texto + 'trece'
      elsif restante == 4
        texto = texto + 'catorce'
      elsif restante == 5
        texto = texto + 'quince'
      elsif restante == 6
        texto = texto + 'dieciséis'
      end

      # Decenas y unidades listas, no tenemos nada restante por escribir.
      restante = 0
    else
      # Los números mayores a 16 pueden escribirse de una forma regular:
      # 'dieci' + unidades
      texto = texto + 'dieci'
    end
  elsif actual == 2 # Grupo del 20 al 29
    # Casos especiales
    if restante == 0
      texto = texto + 'veinte'
    elsif restante == 2
      texto = texto + 'veintidós'
      restante = 0
    elsif restante == 3
      texto = texto + 'veintitrés'
      restante = 0
    elsif restante == 6
      texto = texto + 'veintiséis'
      restante = 0
    else
      # Los demás números del grupo se pueden escribir de forma regular:
      # 'veinti' + unidades
      texto = texto + 'veinti'
```

```
    end
  else # Grupo del 30 al 99
    # Todos estos números se pueden escribir de forma regular:
    # decenas + ' y ' + unidades
    if    actual == 3
      texto = texto + 'treinta'
    elsif actual == 4
      texto = texto + 'cuarenta'
    elsif actual == 5
      texto = texto + 'cincuenta'
    elsif actual == 6
      texto = texto + 'sesenta'
    elsif actual == 7
      texto = texto + 'setenta'
    elsif actual == 8
      texto = texto + 'ochenta'
    elsif actual == 9
      texto = texto + 'noventa'
    end

    if restante > 0
      texto = texto + ' y '
    end
  end
end

actual   = restante # ¿Cuantas unidades quedan por escribir?
restante = 0        # Sustraemos esas unidades

if actual > 0
  if    actual == 1
    texto = texto + 'uno'
  elsif actual == 2
    texto = texto + 'dos'
  elsif actual == 3
    texto = texto + 'tres'
  elsif actual == 4
    texto = texto + 'cuatro'
  elsif actual == 5
    texto = texto + 'cinco'
  elsif actual == 6
    texto = texto + 'seis'
  elsif actual == 7
    texto = texto + 'siete'
  elsif actual == 8
    texto = texto + 'ocho'
  elsif actual == 9
    texto = texto + 'nueve'
  end
end

# Si llegamos hasta aquí, entonces teníamos algún número entre 0 y 100, así
# que debemos retornar la «texto».
```

```
  texto
end

puts numero_a_espanol   0
puts numero_a_espanol   9
puts numero_a_espanol  10
puts numero_a_espanol  11
puts numero_a_espanol  17
puts numero_a_espanol  32
puts numero_a_espanol  88
puts numero_a_espanol  99
puts numero_a_espanol 100
```

```
cero
nueve
diez
once
diecisiete
treinta y dos
ochenta y ocho
noventa y nueve
cien
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Bueno, ciertamente hay algunas cosas que no me gustan de éste programa. Primero, tiene demasiadas repeticiones. Segundo, no maneja números mayores a 100. Tercero, hay demasiados casos especiales, demasiados **`return`**. Usemos algunos arreglos y tratemos de limpiarlo un poco.

```
def numero_a_espanol numero
  if numero < 0  # No aceptamos números negativos.
    return 'Lo siento, el programa no maneja números negativos.'
  end
  if numero == 0
    return 'cero'
  end

  texto = '' # Ésta es la cadena que vamos a devolver.

  # Los arreglos nos permiten manejar los casos especiales de forma ordenada.
  # ¡No más returns!

  unidades = ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis', 'siete', 'ocho',
              'nueve']

  decenas  = ['diez', 'veinte', 'treinta', 'cuarenta', 'cincuenta', 'sesenta',
              'setenta', 'ochenta', 'noventa']

  grupo_10 = ['once', 'doce', 'trece', 'catorce', 'quince', 'dieciséis',
```

```
            'diecisiete', 'dieciocho', 'diecinueve']

grupo_20 = ['veintiuno', 'veintidós', 'veintitrés', 'veinticuatro',
            'veinticinco', 'veintiséis', 'veintisiete', 'veintiocho',
            'veintinueve']

centenas = ['cien', 'doscientos', 'trescientos', 'cuatrocientos', 'quinientos',
            'seiscientos', 'setecientos', 'ochocientos', 'novecientos']

# "restante" es lo que nos falta por escribir del número.
# "actual" es la parte que vamos a escribir ahora.
# "restante" y "actual" ¿Está claro? :)

# Primero obtengamos las centenas...
restante  = numero
actual    = restante / 100         # ¿Cuantas centenas se van a escribir?
restante  = restante - actual * 100 # Sustraemos esas centenas...

if actual > 0
  if actual < 10
    # Como no se usa escribir 'nueve cientos' en lugar de 'novecientos',
    # haremos una excepción para esos casos.
    texto = texto + centenas[actual - 1]
    # El «-1» es debido a que centenas[3] es 'cuatrocientos', no 'trecientos'.
  else
    # Aqui hay un truco que requiere habilidad:
    texto = texto + numero_a_espanol(actual) + ' cientos'
    # Eso es llamado recursión. ¿Qué es lo que acabo de hacer?
    # Le dije a éste método que se llame así mismo, pero con la parte «actual»
    # en lugar de la «restante». Recuerda que la parte «actual» (en éste
    # momento) es el número que tenemos que escribir.
    # Después de que agregamos las «centenas» al «texto», le agregamos la
    # cadena ' cientos'. Así, por ejemplo, si originalmente llamamos a
    # numero_a_espanol con 1999 (de forma que «numero» = 1999), en éste punto
    # lo «actual» sería 19 y lo «restante» sería 99.
    # Lo que implicaría menos esfuerzo en éste punto es dejar a
    # numero_a_espanol escribir 'diecinueve' por nosotros, entonces escribimos
    # ' cientos' y, al final, la siguiente parte de numero_a_espanol escribe
    # 'noventa y nueve'.
  end

  if restante > 0
    # Agregamos «to» a «cien» en caso de que haya más por escribir para los
    # casos donde hay decenas o unidades por escribir, como «ciento cincuenta»
    # y no «cien cincuenta», etc.
    if actual == 1
      texto = texto + 'to'
    end

    # Agregamos un espacio para evitar 'doscientoscincuenta y uno'...
    texto = texto + ' '
  end
end

# Después, obtengamos las decenas...
actual   = restante / 10          # ¿Cuantas decenas se van a escribir?
restante = restante - actual * 10 # Sustraemos esas decenas.

if actual > 0
  if (actual <= 2) and (restante > 0)
```

```
      # Como no se usa escribir 'diez y dos' en lugar de 'doce', haremos
      # una excepción para esos casos.
      if actual == 1
        texto = texto + grupo_10[restante - 1]
      end
      # El «-1» es debido a que grupo_10[3] es 'catorce', no 'trece'.

      # Como no se usa escribir 'veinte y dos' en lugar de 'veintidos', haremos
      # una excepción para esos casos.
      if actual == 2
        texto = texto + grupo_20[restante - 1]
      end
      # El «-1» es debido a que grupo_20[3] es 'veinticuatro', no 'veintitrés'.

      # Como ya nos encargamos del digito en lugar de las unidades, ya no hay
      # nada «restante».
      restante = 0
    else
      texto = texto + decenas[actual - 1]
      # El «-1» es debido a que decenas[3] es 'cuarenta', no 'treinta'.
    end

    if restante > 0
      # Para no escribir 'sesentacuatro'...
      texto = texto + ' y '
    end
  end

  # Por último, si aún queda algo, obtengamos las unidades...
  actual   = restante # ¿Cuantas unidades se van a escribir?
  restante = 0        # Sustraemos esas unidades.

  if actual > 0
    texto = texto + unidades[actual - 1]
    # El «-1» es debido a que unidades[3] es 'cuatro', no 'tres'.
  end

  # Ahora sólo retornamos «texto»...
  texto
end

puts numero_a_espanol(  0)
puts numero_a_espanol(  9)
puts numero_a_espanol( 10)
puts numero_a_espanol( 11)
puts numero_a_espanol( 17)
puts numero_a_espanol( 32)
puts numero_a_espanol( 88)
puts numero_a_espanol( 99)
puts numero_a_espanol(100)
puts numero_a_espanol(101)
puts numero_a_espanol(234)
puts numero_a_espanol(32111)
puts numero_a_espanol(99009)
puts numero_a_espanol(1000000000000)
```

```
cero
nueve
```

```
diez
once
diecisiete
treinta y dos
ochenta y ocho
noventa y nueve
cien
ciento uno
doscientos treinta y cuatro
trescientos veintiuno cientos once
novecientos noventa cientos nueve
cien cientos cientos cientos cientos cientos
```

Prueba el código en línea (da click en «ideone it!» para ejecutar el programa)

Ahhh... Eso es mucho, mucho mejor. El programa es bastante denso, que es por lo cual puse tantos comentarios. Hasta funciona para números grandes... pero no tan bien como uno esperaría. Por ejemplo, pienso que `'un billón'` se vería mejor como el valor retornado para el último número, o hasta `'un millón millón'`. De hecho, podrías hacerlo ahora mismo...

Algunas cosas para intentar

- Expande **`numero_a_espanol`**. Primero, agrega los miles. Debería retornar `'un mil'` (o `'mil'` o hasta `'uno mil'`) en lugar de `'diez cientos'` y `'diez mil'` en lugar de `'cien cientos cientos'`.

- Expande **`numero_a_espanol`** una vez más. Haz que ahora pueda manejar millones para obtener `'un millón'` en lugar de `'un mil mil'`. Después intenta agregar millardos, billones, trillones... ¿Hasta dónde puedes llegar?

- ¿Qué tal un programa que escriba números en estilo de *números de boda*? Debería ser casi lo mismo que **`numero_a_espanol`**, pero insertando «y» entre los grupos de unidades, retornando cosas como `'diecinueve cientos y setenta y dos'` o como se supone que se deben escribir los números en las invitaciones de boda. Te daría más ejemplos, pero yo mismo no lo he terminado de entender. Podrías necesitar contactar a tu organizador de bodas más cercano.

- *«Un elefante se balanceaba...»* Usando **`numero_a_espanol`** y tu viejo programa, escribe la letra de la canción de la forma correcta ésta vez. Castiga a tu computadora: deja que alcance hasta 9,999 elefantes. (Pero tampoco escojas un número muy grande ya que escribir todo eso en la pantalla puede tomar bastante tiempo aún a una computadora; si pones como límite un millón de elefantes, ¡te estarás castigando también a ti mismo!)

¡Felicidades! ¡En éste punto ya eres un verdadero programador! Has aprendido todo lo que necesitas para construir programas enormes desde el fundamento. Si tienes ideas para programas que quisieras escribir para ti mismo, ¡intentalo!

Por supuesto, construir todo desde abajo puede ser un proceso bastante lento. ¿Por qué perder tiempo escribiendo código que alguien más ya escribió? ¿Quisieras que uno de tus programas envíe un correo electrónico? ¿Quisieras guardar y cargar programas en tu computadora? ¿Qué tal generar las páginas web de un tutorial donde todo el código de los ejemplos es probado automáticamente? :) Ruby tiene muchos y diferentes tipos de objetos que podemos usar para ayudarnos a escribir mejores programas aún más rápido.

10. Clases

Hasta ahora, hemos visto diferentes tipos, o *clases*, de objetos: cadenas, enteros, flotantes, arreglos, y algunos cuantos objetos especiales (`true`, `false` y `nil`) de los cuales hablaremos más adelante. En Ruby, éstas clases siempre inician con mayúscula: **`String`**, **`Integer`**, **`Float`**, **`Array`**, etc. En general, si queremos crear un nuevo objeto de alguna clase, usamos **`new`**:

```
a = Array.new + [12345] # Suma de arreglos.
b = String.new + 'hola' # Suma de cadenas.
c = Time.new

puts 'a = ' + a.to_s
puts 'b = ' + b.to_s
puts 'c = ' + c.to_s
```

```
a = 12345
b = hola
c = 2013-10-08 02:37:29 -0500
```

Prueba el código en línea (da click en «ideone it!» para ejecutar el programa)

Debido a que podemos crear arreglas y cadenas usando `[...]` y `'...'`, respectivamente, rara vez los creamos usando **`new`**. (Aunque no es realmente obvio en el ejemplo anterior, **`String.new`** crea una cadena vacía, mientras que **`Array.new`** crea un arreglo vacío.) Los números son excepciones especiales: no puedes crear un entero con **`Integer.new`**. Sólo tienes que escribir el entero.

La clase `Time`

Así que, ¿cuál es la historia con la clase **`Time`**? Los objetos **`Time`** representan momentos en el tiempo. Puedes añadir (o sustraer) números a (o desde) tiempos para obtener nuevos tiempos: añadir `1.5` a un tiempo crea un nuevo tiempo con un segundo y medio después...

```
tiempo1 = Time.new     # El momento en que se ejecuta ésta instrucción.
tiempo2 = tiempo1 + 60 # Un minuto después.

puts tiempo1
```

```
puts tiempo2
```

```
2013-10-08 03:02:48 -0500
2013-10-08 03:03:48 -0500
```

Prueba el código en línea (da click en «ideone it!» para ejecutar el programa)

También puedes crear tiempos para momentos específicos usando **`Time.mktime`**:

```
puts Time.mktime(2000, 1, 1)         # El momento que inició el año 2000.
puts Time.mktime(1976, 8, 3, 10, 11) # El momento en que nací.
```

```
2000-01-01 00:00:00 -0600
1976-08-03 10:11:00 -0600
```

Prueba el código en línea (da click en «ideone it!» para ejecutar el programa)

Nota: Estos tiempos varían dependiendo del tiempo configurado en el reloj de la computadora donde se ejecute el código (él último dígito indica el huso horario que está utilizando). Los paréntesis son para agrupar los parámetros que pasamos a **`mktime`**. Entre más parámetros añadas, más exacto será el tiempo creado.

Puedes comparar tiempos usando los métodos de comparación (un tiempo es *menor que* un tiempo posterior), y si sustraes un tiempo de otro, obtendrás el número de segundos entre ellos. ¡Juega con ello!

Algunas cosas para intentar

- Un billón de segundos... Encuentra el segundo exacto en el que naciste (si es que puedes). Intenta averiguar el segundo exacto en el que tendrás (¿o en el que cumpliste, quizá?) un billón de segundos de edad. Cuando lo averigües, marca la fecha en tu calendario.
- ¡Feliz cumpleaños! Pregunta en qué año nació una persona, después el mes y por último el día. Averigua su edad y dales una **`¡NALGADA!`** Por cada cumpleaños que han tenido.

La clase `Hash`

Otra clase útil es **`Hash`**. Los hashes son muy parecidos a los arreglos, tienen un montón de casillas que pueden apuntar a varios objetos. Sin embargo, en un arreglo, las

casillas están alineadas en una fila y cada una de ellas está numerada (comenzando de cero). En un hash, las casillas no están en una fila (sólo están algo así como juntas en un montón), y puedes usar *cualquier* objeto para hacer referencia a una casilla, no sólo números. Es bueno utilizar hashes cuando quieres tener un registro de un montón de cosas, pero no es necesario tenerlas en una lista ordenada. Por ejemplo, los colores que uso para las diferentes partes de éste tutorial:

```
arregloDeColores = [] # lo mismo que Array.new
hashDeColores    = {} # lo mismo que Hash.new

arregloDeColores[0] = 'rojo'
arregloDeColores[1] = 'verde'
arregloDeColores[2] = 'azul'
hashDeColores['cadenas']    = 'rojo'
hashDeColores['números']    = 'verde'
hashDeColores['reservadas'] = 'azul'

arregloDeColores.each do |color|
  puts color
end

hashDeColores.each do |tipoDeCodigo, color|
  puts tipoDeCodigo + ': ' + color
end
```

```
rojo
verde
azul
cadenas: rojo
reservadas: azul
números: verde
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Si utilizo un arreglo, tengo que recordar que la casilla `0` es para cadenas, la casilla `1` para números, etc. Pero si uso un hash, ¡es fácil! La casilla `'cadenas'` contiene el color de las cadenas de caracteres, claro. Nada que recordar. Te puedes haber dado cuenta que cuando usé **`each`**, los objetos del hash no aparecieron en el mismo orden en el que los colocamos dentro. Los arreglos son para mantener cosas en orden, los hashes no.

Aunque muchas personas usualmente usan cadenas para nombrar las casillas en

un hash, podrías usar cualquier tipo de objeto, aún arreglos u otros hashes (aunque no puedo pensar en porque alguien quisiera hacer esto...):

```
hashRaro = Hash.new

hashRaro[12] = 'monos'
hashRaro[[]] = 'vacío'
hashRaro[Time.new] = 'no hay momento como el presente'
```

Los hashes y arreglos son buenos para diferentes cosas, depende de ti el decidir cuál es mejor para un problema en particular.

Extendiendo clases

Al final del capítulo anterior, escribiste un método para devolver una frase en español cuando recibe un entero, aunque, fue sólo un método genérico de «programa». ¿No sería mucho mejor si pudieras escribir algo como `22.to_esp` en lugar de `numero_a_espanol 22`? Podrías hacerlo de forma parecida a esto:

```
class Integer

  def to_esp
    if self == 5
      espanol = 'cinco'
    else
      espanol = 'cincuenta y ocho'
    end

    espanol
  end
end

# Será mejor que lo pruebe en un par de números...
puts 5.to_esp
puts 58.to_esp
```

```
cinco
cincuenta y ocho
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Bueno, por la prueba, parece funcionar. :)

Saltamos dentro de la clase **`Integer`**, definiendo el método ahí, y saltamos de regreso afuera. Ahora todos los enteros tienen éste método (aunque algo incompleto). De hecho, si no te gustó la forma en que funciona un método pre-construido como **`to_s`**, podrías redefinirlo de la misma forma... pero no te lo recomiendo. Es mejor dejar a los métodos viejos en paz y crear nuevos cuando necesites hacer algo nuevo.

¿Te encuentras confundido? Revisemos un poco más ese último programa. Hasta ahora, cuando ejecutamos cualquier código o definimos cualquier método, lo hicimos dentro del objeto «programa» por defecto. En el programa anterior, dejamos ese objeto por primera vez y nos adentramos en la clase **`Integer`**. Definimos el método ahí (convirtiéndolo en un método de enteros) y todos los enteros pudieron usarlo. Dentro de ese método usamos **`self`** para referirnos al objeto (el entero) que usa el método.

Creando clases

Hemos visto una variedad de diferentes clases de objetos. Sin embargo, es fácil notar los tipos de objetos con los que Ruby no cuenta. Afortunadamente, crear una clase nueva es tan fácil como extender una vieja. Digamos que queremos crear algunos dados en Ruby. Así es como podemos crear la clase **`Dado`**:

```
class Dado

  def rodar
    1 + rand(6)
  end

end

# Creemos un par de dados...
dados = [Dado.new, Dado.new]

# ...y probemos una tirada.
dados.each do |dado|
  puts dado.rodar
end
```

```
3
4
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

(Si te saltaste la sección sobre números aleatorios, **`rand(6)`** nos da un número al azar entre `0` y `5`.)

¡Y eso es! Tus propios objetos.

Podemos escribir todo tipo de métodos para nuestros objetos... pero falta algo. Trabajar con estos objetos se siente mucho como programar antes de que aprendiéramos sobre las variables. Observa nuestro dado, por ejemplo. Podemos hacer que ruede y cada vez nos dará un número diferente, pero si quisiéramos mantener ese número tendríamos que crear una variable que apunte a ese número. Pareciera que cualquier dado decente debería ser capaz de *tener* un número y que **`rodar`** el dado debería cambiar ese número. Si guardamos un registro del dado, ¿no deberíamos tener que guardar un registro del número que está mostrando?

Sin embargo, si tratamos de almacenar el número obtenido al **`rodar`** el dado en la variable local dentro de **`rodar`**, desaparecerá en cuanto **`rodar`** haya terminado. Debemos almacenar el número en un tipo diferente de variable:

Variables de instancia

Normalmente, cuando queremos hablar sobre una cadena de caracteres, simplemente la llamamos una *cadena*. Sin embargo, también podemos llamarla un *objeto cadena*. Algunas veces los programadores pueden llamarla una *instancia de la clase* ***`String`***, pero es sólo una manera formal (y con muchas palabras) de decir que se trata de una *cadena*. Una instancia de una clase, es sólo un objeto de esa clase.

Así que las variables de instancia son sólo las variables de un objeto en particular. Las variables locales de un método viven hasta que el método termina. En cambio, las variable de instancia de un objeto, tendrán la misma vida que el objeto (desaparecerán hasta que el objeto desaparezca). Para diferenciar las variables de instancia de las variables locales, las precedemos con `@` frente a sus nombres:

```
class Dado

  def rodar
    @numeroMostrado = 1 + rand(6)
  end

  def cara
    @numeroMostrado
  end
```

```
end

dado = Dado.new
dado.rodar
puts dado.cara
puts dado.cara
dado.rodar
puts dado.cara
puts dado.cara
```

```
1
1
5
5
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Muy bien! Entonces tiramos el **`dado`** con **`rodar`**, mientras que **`cara`** nos dice que número está mostrando. Sin embargo, ¿qué pasa si tratamos de ver qué número está mostrando antes de rodar el dado (antes de asignar algún valor a **`@numeroMostrado`**)?

```
class Dado

  def rodar
    @numeroMostrado = 1 + rand(6)
  end

  def cara
    @numeroMostrado
  end

end

# Como no voy a volver a usar éste dado otra vez,
# no necesito guardarlo en una variable.
puts Dado.new.cara
```

```
nil
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Hmmm... bueno, al menos no produjo un error. Aun así, no tiene mucho sentido que un dado esté «sin rodar» o lo que sea que `nil` signifique aquí. Sería genial si pudiéramos configurar nuestro nuevo dado justo en el momento en que es creado. Y eso es para lo que sirve **`initialize`**:

```
class Dado

  def initialize
    # Sólo tiraré el dado, aunque también podríamos hacer
    # otra cosa si quisiéramos, como hacer que el dado muestre 6
    rodar
  end

  def rodar
    @numeroMostrado = 1 + rand(6)
  end

  def cara
    @numeroMostrado
  end

end

puts Dado.new.cara
```

```
6
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Cuando un objeto es creado, el método **`initialize`** (si ha sido definido) siempre es llamado.

Nuestro dado es casi perfecto. La única cosa que podría faltar es una forma de ajustar que cara mostrar... ¿por qué no escribes un método **`trampa`** que haga justo eso? Regresa cuando hayas terminado (y que hayas probado que funciona, claro). ¡Asegúrate de que nadie pueda hacer que el dado muestre un `7`!

Hemos cubierto cosas muy interesantes. Aunque tiene su maña, así que permite que dé un ejemplo más interesante. Digamos que queremos crear una mascota virtual sencilla, un pequeño dragón. Como la mayoría de los pequeños, debe ser capaz de

comer, dormir y... dejar uno que otro regalo por ahí, lo cual significa que necesitará que seamos capaces de alimentarlo, llevarlo a la cama y sacarlo a pasear. Internamente, nuestro dragón necesitará llevar un registro de si está hambriento, cansado o debe hacer sus necesidades, pero no seremos capaces de ver eso cuando interactuamos con nuestro dragón, justo como cuando le preguntamos a un bebé humano «¿Tienes hambre?». También añadiremos algunas otras divertidas formas de interaccionar con nuestro pequeño dragón y cuando haya nacido, le daremos un nombre. (Cualquier cosa que pases al método new será pasada al método **`initialize`** por ti.) Muy bien, vamos a intentarlo:

```
class Dragon

  def initialize nombre
    @nombre          = nombre
    @durmiendo       = false
    @nivel_estomago  = 10 # Ésta lleno.
    @nivel_intestino =  0 # No necesita ir al baño.

    puts '¡' + @nombre + ' ha nacido!'
  end

  def alimentar
    puts 'Alimentas a ' + @nombre + '.'
    @nivel_estomago = 10 # A quedado bien relleno.
    paso_del_tiempo
  end

  def pasear
    puts 'Sacas a ' + @nombre + ' a pasear.'
    @nivel_intestino = 0 # No dejará «sorpresas» en la casa.
    paso_del_tiempo
  end

  def acostar
    puts 'Llevas a ' + @nombre + ' a la cama.'
    @durmiendo = true

    3.times do
      if @durmiendo
        puts @nombre + ' ronca, llenando la habitación con humo.'
        paso_del_tiempo
      end
    end

    if @durmiendo
```

```
    @durmiendo = false
    @nombre + ' despierta con calma.'
  end

end

def alzar
  puts 'Cargas a ' + @nombre + ' en brazos, alzandolo a lo alto.'
  puts 'Él ríe, quemando tus pestañas.'
  paso_del_tiempo
end

def mecer
  puts 'Meces a ' + @nombre + ' con gentileza.'
  @durmiendo = true
  puts 'Él cierra los ojos un momento...'
  paso_del_tiempo

  if @durmiendo
    puts '...pero despierta en cuanto te detienes.'
  end
end

private

# «private» significa que los métodos definidos aquí son métodos para uso
# interno del objeto. (Puedes alimentar a tu dragón, pero no puedes
# preguntarle si tiene hambre.)

def hambriento?
  # Los nombres de los métodos pueden terminar con «?».
  # Usualmente sólo se nombran así cuando los métodos sólo retornan true o
  # false. De ésta forma:
  @nivel_estomago  <= 2
end

def incomodo?
  @nivel_intestino >= 8
end

def paso_del_tiempo

  if @nivel_estomago > 0
    # Mover comida del estómago al intestino.
    @nivel_estomago  = @nivel_estomago  - 1
    @nivel_intestino = @nivel_intestino + 1
  else
```

```
      # ¡Nuestro dragón muere de hambre!
      if @durmiendo
        @durmiendo = false
        puts '¡' + @nombre + ' se levanta repentinamente!'
      end
      puts '¡' + @nombre + ' muere de hambre! ¡Desesperado, TE DEVORA!'
      exit # Esto termina el programa.
    end

    if @nivel_intestino >= 10
      puts '¡Ooops! ' + @nombre + ' tuvo un accidente...'
      @nivel_intestino = 0
    end

    if hambriento?
      if @durmiendo
        @durmiendo = false
        puts '¡' + @nombre + ' se levanta repentinamente!'
      end
      puts 'A ' + @nombre + ' le gruñe el estómago...'
    end

    if incomodo?
      if @durmiendo
        @durmiendo = false
        puts '¡' + @nombre + ' se levanta repentinamente!'
      end
      puts @nombre + ' se mueve incómodo de un lado a otro.'
      puts 'Sería buena idea sacarlo a pasear...'
    end
  end

end

mascota = Dragon.new 'Norberto'
mascota.alimentar
mascota.alzar
mascota.pasear
mascota.acostar
mascota.mecer
mascota.acostar
mascota.acostar
mascota.acostar
mascota.acostar
```

```
Norberto ha nacido.
```

```
Alimentas a Norberto.
Cargas a Norberto en brazos, levantandolo a lo alto.
Él ríe, quemando tus pestañas.
Sacas a Norberto a pasear.
Llevas a Norberto a la cama.
Norberto ronca, llenando la habitación con humo.
Norberto ronca, llenando la habitación con humo.
Norberto ronca, llenando la habitación con humo.
Meces a Norberto con gentileza.
Él cierra los ojos un momento...
...pero despierta en cuanto te detienes.
Llevas a Norberto a la cama.
Norberto ronca, llenando la habitación con humo.
¡Norberto se levanta repentinamente!
A Norberto le gruñe el estómago...
Llevas a Norberto a la cama.
Norberto ronca, llenando la habitación con humo.
¡Norberto se levanta repentinamente!
A Norberto le gruñe el estomago...
Llevas a Norberto a la cama.
Norberto ronca, llenando la habitación con humo.
¡Norberto se levanta repentinamente!
A Norberto le gruñe el estomago...
Norberto se mueve incómodo de un lado a otro.
Sería buena idea sacarlo a pasear...
Llevas a Norberto a la cama.
Norberto ronca, llenando la habitación con humo.
¡Norberto se levanta repentinamente!
¡Norberto muere de hambre! ¡Desesperado, TE DEVORA!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Ufff! Por supuesto, sería mejor si fuera un programa interactivo, pero puedes encargarte de eso después. Sólo estaba tratando de mostrar las partes directamente relacionadas con crear una nueva clase dragón.

Observamos algunas cosas nuevas en éste ejemplo. La primera es simple: **`exit`** termina el programa justo ahí y en ese momento. La segunda es la palabra **`private`** con la que nos topamos justo a la mitad de la definición de nuestra clase. Pude haberla dejado fuera, pero quería reforzar la idea de que ciertos métodos son cosas que puedes hacer al dragón y otros son cosas que pasan dentro del dragón mismo. Puedes pensar en ello como lo que está «debajo del cofre» de un auto: a menos que seas un mecánico de autos, todo lo que necesitas conocer es donde está el volante, el acelerador y el freno. Un programador podría llamar a eso la *interfaz pública* de tu auto. El cómo la bolsa de aire sabe cuando debe activarse, sin embargo, es interno al auto; el usuario típico (conductor) no necesita saberlo.

De hecho, para un ejemplo un poco más concreto dentro de esa línea, hablemos

de cómo podemos representar un auto en un juego de vídeo (en lo cual estoy trabajando de momento). Primero, deberías decidir cómo quisieras que luzca tu interfaz pública; en otras palabras, ¿cuáles métodos de tus objetos-auto deberían ser capaces de llamar las personas? Bueno, ellos deberían ser capaces de presionar los pedales del acelerador y el freno, pero también deberían poder especificar con que fuerza lo hacen. (Hay una gran diferencia entre presionarlo suavemente y pisarlo hasta el fondo.) También deberían ser capaces de girar el volante, y de nuevo, de indicar con qué fuerza y en qué dirección están girando el volante. Supongo que podríamos ir aún más lejos y agregar el clutch, las luces direccionales, un lanzador de cohetes, un post-quemador, un condensador de flujo, etc... eso depende el tipo de juego que estés haciendo.

Sin embargo, internamente en el objeto-auto necesita pasar mucho más; otras cosas que el auto necesitaría son velocidad, dirección y posición (siendo lo más básico). Esos atributos podrían ser modificados al presionar el acelerador o el freno y dando vuelta al volante, pero el usuario no debería ser capaz de cambiar la posición del auto directamente (eso sería como teleportarse). También podrías querer llevar un registro del desgaste y el daño al vehículo, si tiene aire en el sistema, y así por el estilo. Todo eso sería al interior de tu objeto-auto.

Algunas cosas para intentar

- Crea la clase **`Naranjo`**. Debe tener el método **`altura`** que retorna su altura, y el método **`doceMesesDespues`**, el cual, cuando es llamado, incrementa la edad del árbol por un año. Cada año, el árbol debe crecer a lo alto (lo que tú consideres que debe crecer un naranjo en un año), y después de cierto número de años (de nuevo, a tu criterio) el árbol debe morir. En los primeros años no debe producir naranjas, pero después de un poco debe hacerlo, y creo que los árboles viejos producen más cada año que los árboles jóvenes... conforme tú creas que tiene más sentido. Por supuesto, debes ser capaz de **`contarLasNaranjas`** (retornando el número de naranjas en el árbol), y **`cortarNaranja`** (que reduce el número de **`@naranjas`** en uno y retorna una cadena diciendo que tan deliciosa estaba esa naranja o que no hay más naranjas para cortar éste año). Asegúrate de que todas las naranjas que no sean cortadas en un año, caigan del árbol antes del siguiente año.

- Escribe un programa con el que puedas interactuar con tu pequeño dragón. Debes ser capaz de introducir comandos como **`alimentar`** y **`pasear`**, y que los métodos correspondientes en tu dragón sean llamados. Claro, como lo que tú escribirás será sólo texto, necesitarás tener algún tipo de *despachador de métodos*, donde tu programa revise el texto que ha sido introducido y llame el

método apropiado.

¡Y eso es todo! Aunque, espera un segundo... Aún no te he contado sobre ninguna de las clases para hacer cosas como mandar correos, o guardar y cargar archivos desde tu computadora, o crear ventanas y botones, o mundos en 3D... ¡nada! Bueno, es que hay *tantas clases* que podrías usar que no podría mostrártelas todas; ¡ni siquiera conozco cuales son la mayoría de ellas! Lo que *sí puedo* decirte es donde encontrarás más acerca de aquellas con las que quieres programar. Pero antes de despedirme, hay una característica más de Ruby sobre la que debes conocer, algo que la mayoría de los lenguajes no tiene, pero sin lo cual yo ya no podría vivir: bloques y procs.

11. Bloques y procs

Ésta es sin duda una de las características más interesantes de Ruby. Algunos otros lenguajes tienen ésta propiedad, aunque pueden llamarla de otra forma (como *clausuras*[22]), pero lamentablemente, la mayoría de los lenguajes populares no la tienen.

¿Qué es ésta cosa nueva tan interesante? Es la habilidad de tomar un *bloque* de código (código entre `do` y `end`), envolverlo en un objeto (llamado *proc*), guardarlo en una variable o pasarlo a un método, y ejecutar el código en el bloque cuando lo desees (y más de una vez si así lo deseas). Por si mismo, es más como un método, excepto que no está ligado a ningún objeto (es un objeto), y puedes almacenarlo o manejarlo como harías con cualquier otro objeto. Creo que es hora de un ejemplo:

```
brindis = Proc.new do
  puts '¡Salud!'
end

brindis.call
brindis.call
brindis.call
```

```
¡Salud!
¡Salud!
¡Salud!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Así, creé un proc (creo que se supone es una abreviación para «procedimiento», pero aún más importante, rima con *block*[23]) que contiene el bloque de código, entonces llamé el proc tres veces. Como puedes ver, es muy parecido a un método.

De hecho, se parece más a un método de lo que te he mostrado, porque los bloques pueden recibir parámetros.

```
teGusta = Proc.new do | algoBueno |
  puts '¡Realmente me *gusta* ' + algoBueno + '!'
end
```

[22] También conocidas con el término en inglés *closures*.
[23] «bloque» en inglés.

```
teGusta.call 'el chocolate'
teGusta.call 'Ruby'
```

```
¡Realmente me *gusta* el chocolate!
¡Realmente me *gusta* Ruby!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Bien, ahora vemos lo que son los bloques y procs, y como podemos usarlos, pero ¿cuál es el punto? Bueno, es porque hay algunas cosas que simplemente no puedes hacer con métodos. Particularmente, no puedes pasar métodos a otros métodos (pero puedes pasar procs a métodos), y los métodos no pueden devolver otros métodos (pero sí pueden devolver procs). Esto es debido a que los procs son objetos y los métodos no.

(Por cierto, ¿algo de esto te parece familiar? Sí, has visto bloques antes... cuando aprendiste sobre iteradores, pero hablemos sobre eso un poco más adelante.)

Métodos que reciben procs

Cuando pasamos un proc a un método, podemos controlar como y cuantas veces se llama al proc. Por ejemplo, digamos que hay algo que queremos hacer antes o después de que algún código sea ejecutado:

```
def algoImportante algunProc
  puts '¡Todos DETÉNGANSE! Tengo que hacer algo...'
  algunProc.call
  puts 'Muy bien todos, he terminado. Continúen con lo que estaban haciendo.'
end

diHola = Proc.new do
  puts 'hola'
end

diAdios = Proc.new do
  puts 'adiós'
end

algoImportante diHola
algoImportante diAdios
```

```
¡Todos DETÉNGANSE! Tengo que hacer algo...
hola
```

```
Muy bien todos, he terminado. Continúen con lo que estaban haciendo.
¡Todos DETÉNGANSE! Tengo que hacer algo...
adiós
Muy bien todos, he terminado. Continúen con lo que estaban haciendo.
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Tal vez eso no parezca especialmente impresionante... ¡pero lo es! :-) Es demasiado común en el mundo de la programación el tener requerimientos estrictos acerca de *qué* y *cuándo* debe ser hecho algo. Si guardas un archivo, por ejemplo, tienes que abrir el archivo, escribir la información que quieres que tenga y entonces cerrar el archivo. Si olvidas cerrar el archivo Cosas Malas™ pueden pasar. Pero cada vez que quieres guardar o cargar un archivo, tienes que hacer las mismas cosas: abrir el archivo, hacer lo que *realmente* quieres hacer y entonces cerrar el archivo. Es tedioso y fácil de olvidar. En Ruby, salvar (o cargar) archivos funciona de forma similar al código de arriba, así que no te tienes que preocupar acerca de nada excepto por lo que quieres guardar (o cargar). (En el siguiente capítulo te mostraré donde puedes encontrar como hacer cosas como guardar o cargar archivos.)

También puedes escribir métodos que determinarán cuántas veces llamar a un proc o aún *si* acaso deben llamarlo. Aquí hay un método que llamará al proc que recibe cerca de la mitad del tiempo y otro método que lo llamará dos veces:

```
def talVez algunProc
  if rand(2) == 0
    algunProc.call
  end
end

def dosVeces algunProc
  algunProc.call
  algunProc.call
end

guino = Proc.new do
  puts '<guiño>'
end

mirada = Proc.new do
  puts '<mirada>'
end
```

```
talVez guino
talVez mirada
dosVeces guino
dosVeces mirada
```

```
<mirada>
<guiño>
<guiño>
<mirada>
<mirada>
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Estos son algunos de los usos más comunes de los procs que nos permiten hacer cosas que no podríamos haber hecho usando sólo métodos. Claro, podrías escribir un método para hacer un guiño dos veces, ¡pero no podrías escribir uno para hacer *algo* dos veces!

Antes de seguir, veamos un último ejemplo. Hasta ahora los procs que hemos pasado han sido bastante similares unos de otros. Ésta vez serán bastante diferentes, así podrás ver como el método depende de los métodos que se le pasan. Nuestro método recibirá algún objeto y un proc, y llamará al proc sobre ese objeto. Si el proc retorna `false`, terminamos; de otra forma, llamamos a proc con el objeto retornado. Continuaremos haciendo esto hasta que el proc retorne `false` (lo cual será mejor que haga eventualmente, o el programa podría fallar). El programa retornará el último valor no falso retornado por el proc.

```
def mientrasNoSeaFalso primeraEntrada, algunProc
  entrada = primeraEntrada
  salida  = primeraEntrada

  while salida
    entrada = salida
    salida  = algunProc.call entrada
  end

  entrada
end

construirArregloDeCuadrados = Proc.new do | arreglo |
  ultimoNumero = arreglo.last
```

```
  if ultimoNumero <= 0
    false
  else
    # Remueve el último numero...
    arreglo.pop
    # ...y reemplazalo con su cuadrado...
    arreglo.push ultimoNumero * ultimoNumero
    # ...seguido por el siguiente número más pequeño.
    arreglo.push ultimoNumero - 1
  end
end

siempreFalso = Proc.new do | soloIgnorame |
    false
end

puts mientrasNoSeaFalso([5], construirArregloDeCuadrados).inspect
puts mientrasNoSeaFalso('Estoy escribiendo esto a las 3:00 am; ¡alguien que
me de un golpe!', siempreFalso)
```

```
[25, 16, 9, 4, 1, 0]
Estoy escribiendo esto a las 3:00 am; ¡alguien que me de un golpe!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Está bien, ese fue un ejemplo bastante raro, lo admito. Pero muestra que tan diferente actúa nuestro método con diferentes procs.

El método **`inspect`** es muy parecido a **`to_s`**, excepto que la cadena que retorna trata de mostrarte el código en Ruby para construir el objeto que recibe. Aquí muestra el arreglo completo retornado por nuestra primera llamada a **`mientrasNoSeaFalso`**. También, te puedes dar cuenta que nunca obtuvimos el cuadrado de ese `0` al final del arreglo, pero como el cuadrado de `0` es aún `0`, no tuvimos que. Y como **`siempreFalso`** fue, tu sabes, siempre **`false`**, **`mientrasNoSeaFalso`** no hizo en realidad nada la segunda vez que lo llamamos; sólo retornó lo que le pasamos.

Métodos que devuelven procs

Otra de las cosas bonitas que puedes hacer con procs es crearlos dentro de métodos y retornarlos. Esto permite todo tipo de locas prácticas de programación (cosas con nombres impresionantes, como *evaluación perezosa*, *estructuras de datos infinitas*, y *currificación*), pero el hecho es que casi nunca lo utilizamos en la práctica, ni puedo recordar haber visto a alguien utilizarlo en su código. Creo que es el tipo de cosas que no tienes que terminar haciendo en Ruby, o tal vez Ruby simplemente alienta a encontrar otras soluciones; no lo sé. En mi caso, sólo hablaré de esto brevemente,

En éste ejemplo, **`componer`** toma dos procs y retorna un nuevo proc que, cuando es llamado, llama al primer proc y pasa el resultado al segundo proc.

```
def componer proc1, proc2
  Proc.new do | x |
    proc2.call(proc1.call(x))
  end
end

alCuadrado = Proc.new do | x |
  x * x
end

alDoble = Proc.new do | x |
  x + x
end

alDobleYalCuadrado = componer alDoble, alCuadrado
alCuadradoYalDoble = componer alCuadrado, alDoble

puts alDobleYalCuadrado.call(5)
puts alCuadradoYalDoble.call(5)
```

```
100
50
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Observa que la llamada a **`proc1`** tuvo que ser dentro de paréntesis al llamar a **`proc2`**, para que **`proc1`** fuera llamado primero.

Pasando bloques (no procs) a métodos

Bien, esto ha sido algo más o menos interesante desde un punto de vista académico, pero es algo poco práctico para usar. Mucho del problema consiste en que hay tres pasos por los que tienes pasar (definir el método, hacer el proc, y llamar el método con el proc), cuando se tiene la sensación de que sólo debería haber dos (definir el método y pasar el *bloque* dentro del método sin tener que usar un proc para nada), ya que la mayor parte del tiempo no querrás usar el bloque/proc después de que lo pases al método. Bueno, podrías no saberlo, ¡pero Ruby tiene todo pensado por nosotros! De hecho, ya lo has estado haciendo cada vez que usas iteradores.

Primero te mostraré un ejemplo y luego hablaremos sobre ello.

```
class Array
  def cadaPar(&eraUnBloque_ahoraUnProc)
    # Comenzamos con «true» porque los arreglos comienzan con 0,
    # que es par.
    esPar = true

    self.each do | objeto |
      if esPar
         eraUnBloque_ahoraUnProc.call objeto
      end

      esPar = (not esPar) # Cambiamos de par a impar, o de impar a par.
    end
  end
end

['manzana', 'manzana podrida', 'cereza', 'durian'].cadaPar do | fruta |
  puts '¡Qué rico! Amo el pay de ' + fruta + '. ¿Tú no?'
end

# Recuerda, estamos obteniendo los elementos pares del arreglo, los que
# resultan ser números impares, sólo porque me gustan problemas así.
[1, 2, 3, 4, 5].cadaPar do | bolaImpar |
  puts '¡' + bolaImpar.to_s + ' NO es un número par!'
end
```

```
¡Qué rico! Amo el pay de manzana. ¿Tú no?
¡Qué rico! Amo el pay de cereza. ¿Tú no?
¡1 NO es un número par!
¡3 NO es un número par!
¡5 NO es un número par!
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

Entonces, para pasar un bloque a **`cadaPar`**, todo lo que tuvimos que hacer fue meter el bloque después del método. Puedes pasar un bloque a cualquier método, aunque muchos métodos simplemente ignorarán el bloque. Para hacer que tu método *no* ignore el bloque, sino que lo tome y lo transforme en un proc, pon el nombre del proc al final de la lista de argumentos del método, precedido por un *et* (**`&`**). Ésta parte tiene su truco, pero no es tan difícil, y sólo tienes que hacer eso una vez (cuando defines el método). Después, puedes usar el método una y otra vez, justo como los métodos pre-construidos que aceptan bloques, como **`each`** y **`times`**. (¿Recuerdas **`5.times do`**...?)

Si estás confundido, sólo recuerda lo que **`cadaPar`** se supone que hace: llamar el bloque que le fue pasado sobre cada tercer elemento alternado del arreglo. Una vez que lo has escrito y funciona, no tienes que pensar acerca de lo que está pasando por debajo («¿qué bloque es llamado cuándo?»); de hecho, eso es exactamente el por qué escribimos métodos como éste: para nunca tener que volver a pensar en cómo funcionan. Sólo los usamos.

Recuerdo que una vez quise ser capaz de tomar la cantidad de tiempo que estaban tomando las diferentes secciones de un programa. (A esto también se le conoce como *análisis de rendimiento* del código o *perfilar* el código.) Así que escribí un método que toma el tiempo antes de ejecutar el código, lo ejecuta, toma el tiempo de nuevo al final y encuentra la diferencia. No puedo encontrar el código ahora mismo, pero lo no necesito; seguramente fue algo como esto:

```
def perfilar descripcionDelBloque, &bloque
  tiempoInicial = Time.now

  bloque.call

  duracion = Time.now - tiempoInicial

  puts descripcionDelBloque + ': ' + duracion.to_s + ' segundos.'
end
```

```
perfilar 'Doblar el valor 25000 veces' do
  numero = 1

  25000.times do
    numero = numero + numero
  end

  # Total de dígitos en éste ENORME número.
  puts numero.to_s.length.to_s + ' dígitos'
end

perfilar 'Contar hasta un millón' do
  numero = 0

  1000000.times do
    numero = numero + 1
  end
end
```

```
752 dígitos
Doblar el valor 25000 veces: 0.246768 segundos
Contar hasta un millón: 0.90245 segundos
```

Prueba el código en línea (da click en ➤ para ejecutar el programa)

¡Qué sencillo! ¡Qué elegante! Con ese pequeño método, ahora puedo con facilidad medir el tiempo de cualquier sección en cualquier programa que quiera hacer; sólo aventaré el programa en un bloque y se lo pasaré a **`perfilar`**. ¿Qué podría ser más simple? En la mayoría de los lenguajes, tendría que añadir explícitamente el código para tomar el tiempo (las cosas en **`perfilar`**) alrededor de cada sección de código que quisiera analizar. En cambio, en Ruby puedo tenerlo todo en un solo lugar y (más importante) ¡fuera de mi camino!

Algunas cosas para intentar

- *El reloj del abuelo*: Escribe un método que reciba un bloque y lo llame por cada hora que ha pasado el día de hoy. De esa forma, si yo le pasara el bloque **`do puts '¡DONG!' end`**, repicaría como un reloj de péndulo. Prueba tu método con unos cuantos bloques diferentes (incluyendo el que acabo de darte). ***Pista:*** *Puedes usar* **`Time.now.hour`** *para obtener la hora actual. Pero esto devuelve*

un número entre `0` *y* `23`*, así que tienes que alterar esos números para obtener números comunes en la carátula de un reloj (del* `1` *a* `12`*).*

- *Registro de programas.* Escribe un método llamado **`registro`**, que toma una cadena como descripción de un bloque y, por supuesto, un bloque, además de otra cadena al final, diciendo que ha terminado el bloque. De forma similar a **`algoImportante`**, debe poner una cadena diciendo que ha comenzado el bloque y otra cadena al final indicando que ha terminado el bloque, indicando lo que el bloque retornó. Prueba tu método enviándole un bloque de código. Dentro de ese bloque, coloca *otra* llamada a **`registro`**, pasando otro bloque. (Esto es llamado *anidar*.) En otras palabras, tu salida debe verse de forma similar a ésta:

```
Comenzando «bloque exterior»...
Comenzando «algún pequeño bloque»...
...«algún pequeño bloque» terminó, retornando: 5
Comenzando «un bloque más»...
...«un bloque más» terminó, retornando: ¡Me gusta la comida tailandesa!
...«bloque exterior» terminó, retornando: false
```

- *Un mejor registro de programas.* La salida del último registro fue un poco difícil de leer y sería peor entre más métodos anidados recibiera. Sería mucho más fácil de leer si sangrara las líneas en los bloques internos. Para hacerlo, necesitarás llevar un registro de que tan profundamente anidado te encuentras cada vez que el registro quiera escribir algo. Para hacerlo, utiliza una *variable global*, una variable que puedes ver desde cualquier parte de tu código. Para crear una variable global, sólo precede el nombre de su variable con **`$`**, como éstas: **`$global`**, **`$profundidadDeAnidado`** y **`$chapulinColorado`**. Al final, tu registro debe presentar algo como esto:

```
Comenzando «bloque exterior»...
    Comenzando «algún pequeño bloque»...
        Comenzando «bloque pequeñito»...
        ...«bloque pequeñito» terminó, retornando: mucho amor
    ...«algún pequeño bloque» terminó, retornando: 42
    Comenzando «un bloque más»...
      ...«un bloque más» terminó, retornando: ¡Me gusta la comida hindú!
...«bloque exterior» terminó, retornando: false
```

Bueno, eso es todo lo que aprenderás en éste tutorial. ¡Felicidades! ¡Has aprendido *muchísimo*! Tal vez no sientes que puedes recordar todo, o te saltaste algunas partes... la verdad, está bien. La programación no es acerca de lo que sabes, es acerca de lo que puedes deducir. Mientras conozcas donde encontrar las cosas que olvidaste, lo harás bien. ¡Espero no creas que escribí todo esto sin buscar cosas a cada minuto! Porque lo hice. También recibí mucha ayuda con el código de los ejemplos en éste tutorial. Pero, ¿dónde estaba buscando cosas y dónde estaba *yo* pidiendo ayuda? Déjame mostrarte...

12. Más allá de éste tutorial

***Nota del traductor**: A continuación, Chris Pine nos presenta tres grandes fuentes de consulta para resolver las dudas sobre Ruby. El único pequeño inconveniente, es que éstas fuentes se encuentran en el idioma inglés (lo cual podría ser una buena motivación para aprenderlo), sin embargo, he recopilado algunas fuentes de consulta disponibles en español que pueden ser de ayuda para quienes no dominen el inglés. Ésta información se encuentra un poco más adelante.*

Así que, ¿a dónde vamos ahora? Si tienes una pregunta, ¿a quién le puedes preguntar? ¿Qué hay si quieres que uno de tus programas abra una página web, envíe un correo o cambie el tamaño de una fotografía digital? Bueno, hay muchos, muchos lugares para encontrar ayuda sobre Ruby. Pero decir sólo eso no es de mucha ayuda, ¿verdad? :-)

Para mí, en realidad sólo hay tres lugares en donde busco por ayuda sobre Ruby. Si se trata de una pequeña pregunta y creo que puedo experimentar por mi cuenta para encontrar la respuesta, uso *irb*. Si es una pregunta más compleja, busco dentro de mi *pickaxe*. Y si no puedo encontrar la respuesta por mí mismo, pido ayuda en *ruby-talk*.

- IRB: Ruby Interactivo

  Si instalaste Ruby, entonces también instalaste irb. Para usarlo, sólo abre tu línea de comandos y escribe `irb`. Cuando estás en irb, puedes escribir cualquier expresión de Ruby que desees y te dirá el valor de la misma. Escribe `1 + 2` y te dirá `3`. (Observa que no tienes que usar `puts`.) Es como una calculadora gigante de Ruby. Cuando hayas terminado, sólo escribe `exit`.

  Hay mucho más sobre irb que esto, pero puedes aprender sobre ello en el «pickaxe».

- El «pickaxe»: Programando en Ruby

  Absolutamente, el libro de Ruby a obtener es «Programming Ruby, The Pragmatic Programmer's Guide», por David Thomas y Andrew Hunt (Los Programadores Pragmáticos). Aunque recomiendo ampliamente conseguir la edición más reciente de éste excelente libro, también puedes conseguir una un poco vieja (pero aún muy relevante) versión gratuita en línea. (De hecho, si instalaste la versión para Windows de Ruby, ya cuentas con la versión gratuita del libro.)

Puedes encontrar casi todo sobre Ruby, desde lo básico a lo avanzado, en éste libro. Es fácil de leer; es detallado; es casi perfecto. Desearía que todos los lenguajes tuvieran un libro de ésta calidad. En la parte trasera del libro, encontrarás una enorme sección detallando cada método en cada clase, explicándolo y dando ejemplos. ¡Simplemente amo éste libro!

Hay varios lugares donde puedes obtenerlo (incluyendo el propio sitio de los Programadores Pragmáticos), pero mi lugar favorito es ruby-doc.org. Esa versión tiene una bonita tabla de contenidos a un lado, como también un índice. (ruby-doc.org cuenta también con mucho más documentación, como la del API central y de la Librería Estándar... básicamente, documenta todo lo trae Ruby consigo al sacarlo de la caja. Revísalo.)

Y, ¿por qué es llamado «pickaxe»? Bueno, hay una imagen de un zapapico en la portada el libro. Es un nombre tonto, me imagino, pero es el que se quedó.

- Ruby-Talk: La lista de correo sobre Ruby

Aún con irb y el «pickaxe», algunas veces no puedes encontrar la respuesta. O tal vez quieres averiguar si alguien ya hizo lo que sea que es en lo que estás trabajando, para ver si puedes optar por usarlo. En esos casos, el lugar para hacerlo es ruby-talk, la lista de correo sobre Ruby. Está llena de personas amigables, inteligentes y dispuestas a ayudar. Para aprender más o suscribirte, visita éste enlace.

ADVERTENCIA: Hay muchos mensajes al día en la lista. Yo tengo mi correo configurado para que los mensajes de ruby-talk se ordenen automáticamente en una carpeta para que no se mezcle con otros mensajes. Si simplemente no deseas lidiar con tanto correo, ¡no tienes por qué hacerlo! La lista de correo ruby-talk es duplicada al grupo de noticias comp.lang.ruby y viceversa, así que puedes leer los mismos mensajes ahí, sólo que en un formato ligeramente diferente.

Recursos en español

Lamentablemente, aún no hay una fuente de documentación oficial para Ruby en español. Sin embargo, te presento enlaces a material disponible en la red donde espero puedas resolver la mayoría tus dudas.

- Un par de textos

En primer lugar tenemos la «Guía del usuario de Ruby». Ésta guía fue

escrita en japonés por el mismísimo creador del lenguaje, Yukihiro Matzumoto, y después traducida y adaptada al inglés por Kentaro y Slagell. Al parecer, un hispanohablante sólo conocido como Paco González realizó la traducción al español. Puedes descargar el archivo, que contiene una presentación general de las funciones del lenguaje.

También puedes encontrar en la red, el texto «Ruby Fácil», de Diego Guillén. Contiene los aspectos básicos del lenguaje, llega a tocar los temas de desarrollo con librerías para web, trabajo con interfaces gráficas, conexiones a bases de datos, etc. Puedes encontrar una versión antigua en la red, aunque en su versión más reciente y de contenido ampliado no es gratuito, pero es de lo poco que se puede conseguir en español por el momento.

- Ruby en español (Google+ y Facebook)

  Son dos comunidades de hispanohablantes que se han reunido en torno a éstas redes sociales. Usualmente se comparten enlaces a artículos sobre Ruby y seguramente te ayudarán si tienes alguna duda (aunque la respuesta puede tardar un poco). Sólo asegúrate de leer las reglas de los grupos antes de publicar algo, así evitaras molestar a los moderadores y será más fácil que te proporcionen ayuda. Aquí encontrarás el grupo de Google+, y aquí encontrarás el grupo de Facebook. (Aunque ambos comparten el mismo nombre, son grupos diferentes.) También se encuentra en Google+ el grupo «Ruby y Ruby-on-Rails en México», aunque los temas giran más en torno a Rails (un *framework*[24] para desarrollo web).

- Para reforzar lo aprendido

  Aunque el siguiente par de recursos no es para consulta, sí te pueden ayudar a reforzar lo que has aprendido en éste tutorial (y muy posiblemente aprender una cosa nueva o dos). El primero en el curso en línea de Ruby de CodeAcademy. Por medio de ejemplos y ejercicios donde tienes que ejecutar y comprender lo que hace el código para poder avanzar, te muestran las características generales del lenguaje. Es bastante recomendable, ya que te otorgará bastante práctica escribiendo código.

  El último recurso del que te hablaré aquí es el vídeo-tutorial de introducción a Ruby de DevCode.la. Aquí podrás repasar varios de los conceptos del tutorial viendo cómo funcionan los ejemplos durante la explicación, lo cual podría esclarecer algunos puntos oscuros.

[24] Conjunto de programas que facilitan la construcción de otros programas.

Tim Toady

Algo de lo que he tratado de protegerte, pero con lo que seguramente te encontrarás pronto, es el concepto de TMTOWTDI[25] (pronunciado en inglés «Tim Toady»): hay más de una manera de hacerlo.

Algunas personas te dirán que TMTOWTDI es algo genial, mientras que otros lo ven de una forma distinta. En general no tengo una opinión en contra o a favor, pero si pienso que es una *terrible* forma de enseñarle a alguien como programar. (¡Cómo si aprender una forma de hacer algo no fuera lo suficientemente retador y confuso!)

Sin embargo, ahora que te mueves más allá de éste tutorial, verás código mucho más diverso. Por ejemplo, puedo pensar en por lo menos otras cinco formas de crear una cadena de texto (además de rodear el texto con comillas rectas sencillas), y cada una funciona de una forma ligeramente diferente. Sólo te mostré la más sencilla de las seis.

Y cuando hablamos sobre ramificación, te mostré **`if`**, pero no te mostré **`unless`**. Dejaré que averigües cómo funciona usando irb.

Otro pequeño y atajo que puedes usar con **`if`**, **`unless`** y **`while`**, es la bonita versión de una línea:

```
# Éstas líneas son de un programa que escribí para generar balbuceos
# parecidos al inglés. Cool, ¿verdad?
puts 'grobably combergearl kitatently thememberate' if 5 == 2**2 + 1**1
puts 'enlestrationshifter supposine follutify blace' unless 'Chris'.length == 5
```

```
grobably combergearl kitatently thememberate
```

Y finalmente, hay otra forma de escribir métodos que reciben bloques (no procs). Vimos la cosa donde tomamos el bloque y lo volvimos un proc usando el truco de **`&bloque`** en la lista de parámetros cuando defines la función. Entonces, para llamar el bloque, simplemente usas **`bloque.call`**. Bueno, hay una forma más corta (aunque personalmente la encuentro más confusa). En lugar de esto:

```
def dosVeces(&bloque)
  bloque.call
  bloque.call
end
```

[25] Acrónimo de la frase en inglés «There's More Than One Way To Do It».

```
dosVeces do
  puts 'murditivent flavitemphan siresent litics'
end
```

```
murditivent flavitemphan siresent litics
murditivent flavitemphan siresent litics
```

...haces esto:

```
def dosVeces
  yield
  yield
end

dosVeces do
  puts 'buritiate mustripe lablic acticise'
end
```

```
buritiate mustripe lablic acticise
buritiate mustripe lablic acticise
```

No sé... ¿tú qué piensas? Tal vez soy sólo yo, pero... ¡¿**`yield`**?! Si fuera algo como **`llamar_bloque_oculto`** o algo similar, eso tendría más sentido para mí. Mucha gente dice que **`yield`** tiene sentido para ellos. Pero me imagino que eso es de lo que se trata TMTOWTDI: ellos lo hacen a su manera y yo lo hago en la mía.

El fin

Tal vez te puedes preguntar, ¿en qué puedes usar Ruby y todo lo que has aprendido? Ve el siguiente vídeo donde Mario Chavez habla sobre los usos que les puedes dar a Ruby y deja volar tu imaginación, pero recuerda, usalo para el bien y no para el mal. :-) Y, si encontraste éste tutorial útil (o confuso, o encontraste algún error), ¡déjame saber![26] Además, no dudes en compartirlo con todos a quien creas que les puede interesar aprender a programar en un bonito lenguaje.

[26] De nuevo, recuerda que si envías algún comentario al sr. Pine, es mejor hacerlo en el idioma inglés. En cambio, si tienes un comentario sobre la traducción al español, es mejor contactarme. (También puedes colaborar mediante el repositorio del proyecto.)

13. Soluciones a los problemas propuestos

A continuación te presento una lista de enlaces que llevan al repositorio con las soluciones en código a los problemas encontrados en las secciones tituladas «Algunas cosas para intentar». Cada una de las soluciones presentadas debe consultarse sólo como último recurso para intentar resolver el problema y en el entendido de que la solución presentada es sólo una de muchas posibles.

Si llevas varios días sin poder crear alguno de los programas, revisa el código de la solución, pero asegúrate de leerlo con cuidado y comprender que es lo que hace cada instrucción de forma específica y como ayuda a resolver el problema en general.

Traté de utilizar solamente las partes del lenguaje vistas hasta el momento en la sección correspondiente a cada programa, aunque puede que se me escapara alguna que otra instrucción nueva, pero seguramente podrás deducir o investigar como funciona. También coloqué muchos comentarios en los programas, para tratar de dejar bien claro que es lo que hace cada parte.

- **Números**
    1. ¿Cuántas horas hay en un año?
    2. ¿Cuántos minutos hay en una década?
    3. ¿Cuál es tu edad en segundos?
    4. ¿Cuántos chocolates comerás en toda tu vida (aproximadamente)?
    5. ¿Qué edad tengo?
- **Mezclando todo**
    1. ¡Saludos!
    2. Número favorito.
- **Más acerca de los métodos**
    1. Jefe enojado.
    2. Tabla de contenidos.
- **Control de flujo**
    1. Un elefante se balanceaba.
    2. La abuela sorda.
    3. La abuela necia.
    4. Años bisiestos.
- **Arreglos e iteradores**
    1. Ordenar palabras.
    2. Ordenar palabras sin `sort`.
    3. Nueva tabla de contenidos.

- **Escribiendo tus propios métodos**
    1. Miles de números a español.
    2. Trillones (o más) de números a español.
    3. Números para invitaciones.
    4. 9,999 elefantes se balanceaban.
- **Clases**
    1. Mil millones de segundos.
    2. ¡Feliz cumpleaños!
    3. Árbol de naranjas.
    4. Pequeño dragón.
- **Bloques y procs**
    1. El reloj del abuelo.
    2. Registro de programas.
    3. Un mejor registro de programas.

En caso de que encuentres algún error en los programas o creas que alguno puede escribirse de una forma más sencilla, por favor, clona el repositorio de los programas y manda un «pull request» con los cambios. La alternativa en enviarme un mensaje de correo.

-- David O' Rojo