

LUCAS EGA ACADEMY

Ruby



Curso Completo de Ruby

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

ÍNDICE

PRÓLOGO	6
Bienvenido a tu curso de Ruby	6
Qué es Ruby y por qué aprenderlo	6
A quién va dirigido este libro	7
Cómo está organizado el libro	8
Cómo estudiar este libro	9
Qué necesitas para empezar	10
Un mensaje de Esteban	10
Antes de empezar: tu primer programa	11
La filosofía de este libro: código que se lee como historia	11
Errores típicos de quien empieza (y cómo evitarlos)	12
Lo que vas a poder hacer al terminar	12
Preguntas frecuentes antes de empezar	13
Tu entorno de trabajo: déjalo listo desde hoy	14
UNIDAD 1: Bienvenido a Ruby	15
Explicación detallada	15
Ejemplos paso a paso	29
Práctica	33
Resumen de la unidad	37
UNIDAD 2: Variables y datos	46
Explicación detallada	46
Ejemplos paso a paso	63
Práctica	67
Resumen de la unidad	80
UNIDAD 3: Tomando decisiones	82
Explicación detallada	82
Ejemplos paso a paso	100
Práctica	106
Resumen de la unidad	117
UNIDAD 4: Bucles y arreglos	119

Explicación detallada	119
Ejemplos paso a paso	140
Práctica	147
Resumen de la unidad	152
Ejemplos paso a paso (segunda parte)	158
Práctica (segunda parte)	161
Lo que sigue	166
UNIDAD 5: Hashes y métodos	167
Explicación detallada	167
Ejemplos paso a paso	190
Práctica	197
Resumen de la unidad	204
UNIDAD 6: Bloques y objetos	206
Explicación detallada	206
Ejemplos paso a paso	234
Práctica	243
Resumen de la unidad	254
UNIDAD 7: Herencia, módulos y strings	256
Explicación detallada	256
Ejemplos paso a paso	287
Práctica	302
Resumen de la unidad	312
UNIDAD 8: Errores, archivos y tiempo	314
Explicación detallada	314
Ejemplos paso a paso	332
Práctica	345
Resumen de la unidad	360
UNIDAD 9: Gemas, pruebas y regex	362
Explicación detallada	363
Ejemplos paso a paso	391
Práctica	397
Resumen de la unidad	403

UNIDAD 10: Proyecto final e intro a Rails	405
Explicación detallada	405
Ejemplos paso a paso	431
Práctica	440
Resumen de la unidad	449
EXAMEN FINAL	450
Guía de repaso rápido antes del examen	450
Mapa de temas por pregunta	451
Respuestas y soluciones	458
Cómo calificar tu examen y qué sigue	469
Las 10 trampas más comunes de este examen (y cómo evitarlas)	471

PRÓLOGO

Bienvenido a tu curso de Ruby

¡Hola! Qué bueno que estás aquí. Si llegaste a este libro, es porque quieres aprender a programar con Ruby, y déjame decirte algo desde ahorita: tomaste una excelente decisión. Ruby no es solo un lenguaje de programación, es una forma elegante y feliz de pensar sobre el código.

Lucas EGA Academy existe para una sola razón: enseñarte programación de verdad, en español, gratis, paso a paso, sin vueltas. Este libro es parte de la serie "Curso Completo" de la academia, donde cada curso está diseñado para llevarte de cero a programador competente con práctica real, ejercicios resueltos y exámenes que ponen a prueba lo que de verdad aprendiste.

Este libro no asume que sepas programar. Empieza desde lo más básico - qué es Ruby, cómo instalarlo, cómo escribir tu primer programa - y te lleva hasta temas intermedios como programación orientada a objetos, módulos, manejo de archivos, expresiones regulares y el ecosistema de gemas. Si ya sabes programar en otro lenguaje, perfecto: vas a avanzar rápido y vas a descubrir por qué Ruby tiene fama de ser el lenguaje más elegante que existe.

Qué es Ruby y por qué aprenderlo

Ruby es un lenguaje de programación creado por Yukihiro Matsumoto, conocido como "Matz", en Japón, a mediados de los años noventa. Matz tenía una idea muy clara cuando lo creó: quería un lenguaje que hiciera felices a los programadores. Literalmente, esa fue su filosofía de diseño: "Quiero que programar sea divertido". Y vaya que lo logró.

Ruby es un lenguaje interpretado, dinámico y orientado a objetos. Eso significa tres cosas importantes para ti. Primero, interpretado: no necesitas compilar nada, escribes el código y lo corres, así de directo. Segundo, dinámico: no tienes que declarar tipos de variables, el lenguaje se encarga de eso por ti. Tercero, orientado a objetos: todo en Ruby es un objeto, hasta los números y los textos tienen métodos. Eso hace que el código se lea casi como inglés (o como español traducido) y que escribir programas se sienta natural.

Pero, ¿por qué aprender Ruby hoy? Te doy las razones de peso:

Primero, Ruby on Rails. Rails es el framework web construido con Ruby que cambió la forma de hacer aplicaciones web. Empresas como GitHub, Shopify, Airbnb (en sus inicios), Basecamp y miles de startups usan o usaron Rails. Con Ruby y Rails puedes construir una aplicación web completa - con base de datos, usuarios, pagos - en días, no en meses. Es el lenguaje favorito de los emprendedores que quieren lanzar rápido.

Segundo, scripting y automatización. Ruby es buenísimo para escribir scripts que te ahorran horas de trabajo repetitivo: renombrar mil archivos, procesar textos, descargar información de internet, automatizar tareas del sistema. Los administradores de sistemas y los analistas de datos lo usan todos los días para eso.

Tercero, la elegancia. En serio, el código Ruby es bonito. Mira la diferencia: en otros lenguajes necesitas varias líneas de código ceremonial para hacer algo simple; en Ruby lo haces en una línea que se lee sola. Por ejemplo, para imprimir los números del 1 al 5, en Ruby escribes `(1..5).each { |n| puts n }`. Corto, claro, directo. Esa elegancia no es vanidad: el código que se lee fácil se mantiene fácil, y eso vale oro en el trabajo real.

Cuarto, la comunidad. La comunidad Ruby es famosa por ser amable y por producir herramientas de gran calidad. Las "gemas" (gems) son paquetes de código que otros programadores comparten gratis: hay gemas para casi todo lo que se te ocurra. Además, la cultura de Ruby valora las buenas prácticas, las pruebas automatizadas y el código limpio, cosas que te van a servir en cualquier lenguaje.

Quinto, es un gran primer lenguaje... y un gran segundo lenguaje. Si es tu primero, Ruby te enseña los conceptos fundamentales (variables, condicionales, bucles, funciones, objetos) sin pelearte con sintaxis complicada. Si ya programas en Python, JavaScript o Java, Ruby te va a abrir la cabeza a nuevas formas de resolver problemas, como los bloques y la metaprogramación.

A quién va dirigido este libro

Este libro es para ti si:

- Nunca has programado y quieres empezar con un lenguaje amable y poderoso.
- Ya programas en otro lenguaje y quieres agregar Ruby a tu caja de herramientas.
- Quieres aprender Ruby on Rails y necesitas primero dominar el lenguaje base.
- Te interesa la automatización, el scripting o el desarrollo web.
- Quieres entender la programación orientada a objetos de verdad, no solo la teoría.

No es para ti si buscas un manual de referencia seco o si quieres aprender Rails directamente sin pasar por Ruby (créeme, primero domina el lenguaje; Rails sin Ruby es como querer correr sin saber caminar).

Cada capítulo asume que leíste los anteriores, pero si ya tienes experiencia puedes saltar las primeras unidades sin problema. Lo importante es que hagas los ejercicios: leer sobre programar no te hace programador, programar te hace programador.

Cómo está organizado el libro

El libro está dividido en 10 unidades, cada una enfocada en un tema grande. La progresión está pensada para que cada unidad construya sobre la anterior:

Unidad 1: Bienvenido a Ruby. Qué es Ruby, cómo instalarlo, tu primer programa y la consola interactiva irb.

Unidad 2: Variables y datos. Variables, tipos de datos y entrada y salida básica: la materia prima de todo programa.

Unidad 3: Tomando decisiones. Condicionales (if, unless, case) y cómo tu código decide qué camino tomar.

Unidad 4: Bucles y arreglos. Bucles (while, until, each, times) y arreglos: repetir tareas y guardar listas de datos.

Unidad 5: Hashes y métodos. Pares clave-valor con hashes y cómo organizar tu código en métodos reutilizables.

Unidad 6: Bloques y objetos. El concepto estrella de Ruby (bloques, procs y lambdas) y la programación orientada a objetos: clases, objetos y encapsulamiento.

Unidad 7: Herencia, módulos y strings. Reutilizar código con herencia y módulos (mixins con include y extend), y manipulación avanzada de strings.

Unidad 8: Errores, archivos y tiempo. Excepciones con begin/rescue, lectura y escritura de archivos, y manejo de fechas y tiempo.

Unidad 9: Gemas, pruebas y regex. El ecosistema de gemas y Bundler, pruebas básicas y el poder de las expresiones regulares.

Unidad 10: Proyecto final e intro a Rails. Un proyecto integrador con todo lo aprendido y tu primera mirada a Ruby on Rails.

Cada unidad incluye explicaciones con ejemplos de código que puedes correr, ejercicios prácticos al final y un resumen de los puntos clave. Al final del libro encontrarás el examen final de 50 preguntas que cubre todo el contenido: si lo pasas con buena nota, puedes decir con confianza que dominas los fundamentos de Ruby.

Cómo estudiar este libro

Te voy a dar los consejos que les doy a todos mis estudiantes, porque la diferencia entre quien aprende y quien no, casi nunca es la inteligencia: es el método.

Primero, programa todos los días, aunque sea 30 minutos. La programación es una habilidad, como tocar un instrumento: la constancia le gana al talento. Mejor 30 minutos diarios que 5 horas un domingo al mes.

Segundo, escribe el código tú mismo, no lo copies y pegues. Cuando veas un ejemplo en el libro, escríbelo a mano en tu editor, córrelo, y luego modifícalo: cambia valores, rompe cosas a propósito, mira qué errores salen. Los errores son tus mejores maestros.

Tercero, usa irb como tu laboratorio. irb es la consola interactiva de Ruby: la abres, escribes código y ves el resultado al instante. ¿No estás seguro de qué hace un método? Pruébalo en irb. ¿Quieres experimentar con un arreglo? Hazlo en irb. Es la herramienta más subestimada por los principiantes y la más usada por los profesionales.

Cuarto, haz todos los ejercicios, sin excepción. Los ejercicios están diseñados para que te atores un poco: esa incomodidad es donde ocurre el aprendizaje. Si un ejercicio te sale a la primera, bien; si te cuesta, mejor, porque significa que estás estirando tu capacidad. Y cuando termines, compara tu solución con la propuesta, pero no antes de intentarlo de verdad.

Quinto, construye tus propios mini proyectos. Después de cada dos o tres unidades, inventa un programita que use lo aprendido: una calculadora, un juego de adivinanzas, un organizador de tareas en la terminal. Los proyectos propios son lo que convierte el conocimiento en habilidad.

Sexto, no te saltes lo que no entiendas. Si un tema se te complica, relee la unidad, busca otro ejemplo, pregúntale a alguien o búscalo en internet. Los temas de Ruby se encadenan: si dejas un hueco en métodos, los bloques te van a costar el doble.

Séptimo, enseña lo que aprendes. Explícale a un amigo (o a ti mismo en voz alta) cómo funciona un bloque o qué es la herencia. Si puedes explicarlo con palabras simples, es que lo entendiste de verdad.

Qué necesitas para empezar

La buena noticia: necesitas muy poco.

Una computadora. Cualquier sistema operativo sirve: Windows, macOS o Linux. Ruby funciona en los tres. En el libro te explico cómo instalarlo en cada uno, pero te adelanto que en macOS y Linux suele venir preinstalado o se instala con un comando, y en Windows hay un instalador gráfico muy sencillo llamado RubyInstaller.

Un editor de texto o IDE. Puedes empezar con algo simple como Visual Studio Code (gratis y muy popular), y más adelante probar editores hechos para Ruby. No necesitas nada sofisticado al principio.

Conexión a internet. Para descargar Ruby, consultar la documentación oficial y buscar ayuda cuando te atores. La documentación oficial de Ruby (ruby-doc.org) va a ser tu mejor amiga.

Ganas de practicar. Esto no se compra ni se descarga, pero es lo más importante de la lista.

No necesitas saber matemáticas avanzadas, no necesitas haber programado antes, no necesitas una computadora potente. Solo necesitas empezar.

Un mensaje de Esteban

Hola, soy Esteban, el fundador de Lucas EGA Academy. Quiero contarte por qué existe este libro.

Cuando empecé a aprender programación, me topé con el mismo problema una y otra vez: los recursos buenos estaban en inglés, y los que estaban en español eran pocos, caros o desactualizados. Yo quería aprender, tenía las ganas, pero me faltaba alguien que me explicara las cosas paso a paso, en mi idioma, sin asumir que yo ya sabía la mitad.

Por eso creé Lucas EGA Academy: una academia en línea, en español, con cursos gratuitos de tecnología. Y por eso existen estos libros: para que tengas en tus manos (o en tu pantalla) un curso completo, ordenado y práctico, que te lleve de cero a programador sin que tengas que pagar un peso.

Ruby tiene un lugar especial para mí porque fue uno de los lenguajes que me enseñó que programar puede ser elegante y divertido. Cuando escribes Ruby bien, el código se siente como poesía lógica: corto, expresivo, bonito. Quiero que tú sientas eso también.

Pero te voy a ser directo: este libro no va a programar por ti. Yo te doy el mapa, los ejemplos y los ejercicios, pero el que camina eres tú. Va a haber momentos en que un ejercicio no te

salga, en que un error te frustre, en que quieras cerrar la computadora. En esos momentos, acuérdate de esto: todos los programadores pasamos por ahí. La diferencia entre un programador y alguien que "intentó programar" es que el programador no se rindió ese día.

Así que échale ganas, practica todos los días, termina el libro, pasa el examen final y luego sigue con Rails o con lo que tú quieras. El conocimiento que vas a ganar aquí nadie te lo puede quitar.

Nos vemos en la meta. ¡A darle!

Antes de empezar: tu primer programa

Para que llegues motivado a la Unidad 1, te dejo un adelanto. Cuando instales Ruby, vas a poder escribir tu primer programa. Va a ser algo así:

```
puts "Hola, mundo. Voy a aprender Ruby."
```

Esa línea, cuando la corres, le dice a la computadora que muestre ese mensaje en la pantalla. Simple, ¿verdad? Pues todo lo que vas a aprender en este libro son formas más poderosas de darle instrucciones a la computadora: guardar datos, tomar decisiones, repetir tareas, organizar código y construir tus propias herramientas.

Para el final del libro, ese "Hola, mundo" se va a haber convertido en programas que leen archivos, manejan errores, definen sus propias clases y hasta juegan contigo. Todo paso a paso. Todo en español. Todo gratis.

Bienvenido a Ruby. Bienvenido a Lucas EGA Academy. Empecemos.

La filosofía de este libro: código que se lee como historia

Hay una frase famosa en la comunidad Ruby, escrita por el propio Matz, que dice algo así: "Ruby es simple en apariencia, pero es muy complejo por dentro, como el cuerpo humano". Lo que quiso decir es que Ruby esconde la complejidad para que tú no tengas que pelearte con ella. Este libro sigue esa misma filosofía: las explicaciones van a ser simples en apariencia, pero cada ejemplo está elegido para enseñarte algo profundo.

Vas a notar que en este libro el código siempre viene con contexto. No te voy a soltar un fragmento abstracto y esperar que lo entiendas: te voy a decir qué problema resuelve, por qué se escribe así y qué pasaría si lo escribieras de otra forma. Porque programar no es memorizar sintaxis, es aprender a pensar en soluciones.

También vas a notar que repito mucho una idea: en Ruby hay casi siempre más de una forma de hacer las cosas. ``puts "hola" y `puts("hola")`` hacen lo mismo. Puedes escribir un ``if`` en una línea o en varias. Eso no es un defecto del lenguaje, es una característica: Ruby confía en ti y te da libertad. Pero con la libertad viene la responsabilidad de escribir código que otros (incluido tu yo del futuro) puedan leer. A lo largo del libro te voy a enseñar las convenciones de la comunidad para que tu código no solo funcione, sino que se vea profesional.

Errores típicos de quien empieza (y cómo evitarlos)

Te adelanto los tropiezos más comunes para que no te asusten cuando lleguen:

Querer memorizar en lugar de entender. No memorices métodos; entiende los patrones. Si entiendes cómo funcionan los bloques, vas a intuir cómo usar ``each``, ``map`` y ``select`` sin memorizar cada uno.

No leer los mensajes de error. Los errores de Ruby son tus amigos: te dicen exactamente qué línea falló y por qué. Acostúmbrate a leerlos completos en lugar de entrar en pánico.

Copiar código sin escribirlo. Ya lo dije arriba, pero lo repito porque es el error número uno: el código que solo lees se olvida; el código que escribes se queda.

Saltarse los ejercicios "fáciles". Los ejercicios fáciles construyen la base para los difíciles. Si los saltas, los difíciles te van a parecer imposibles y no es que sean imposibles: es que te faltó la base.

Compararse con otros. Alguien siempre va a avanzar más rápido que tú. No importa. Lo único que importa es que hoy sepas un poco más que ayer.

Lo que vas a poder hacer al terminar

Cuando termines este libro y apruebes el examen final, vas a poder:

- Escribir programas en Ruby desde cero, con buena estructura y estilo.
- Usar arreglos, hashes, strings y números con soltura.
- Crear tus propias clases, usar herencia y módulos como un profesional.
- Manejar errores de forma elegante para que tus programas no se rompan.
- Leer y escribir archivos, procesar texto y usar expresiones regulares.
- Instalar y usar gemas, y entender el ecosistema Ruby.

- Leer código Ruby de otras personas y entender qué hace.
- Dar el salto a Ruby on Rails con una base sólida del lenguaje.

Eso no es poca cosa: es exactamente lo que piden los trabajos junior de desarrollo Ruby. Y si tu meta no es trabajar de programador sino automatizar tus tareas o construir tus propios proyectos, también vas a tener todo lo necesario.

El camino es largo pero está bien señalado. Cada unidad termina donde empieza la siguiente, cada ejercicio te prepara para el siguiente tema, y el examen final te va a demostrar cuánto avanzaste. Confía en el proceso, haz tu parte diaria, y nos vemos al final convertidos en programadores Ruby.

¡Vámonos!

Preguntas frecuentes antes de empezar

Antes de entrarle de lleno, respondo las preguntas que más me hacen los estudiantes:

¿Necesito saber inglés? No es obligatorio, pero te ayuda. La mayoría de la documentación, los mensajes de error y los nombres de los métodos están en inglés. Con un inglés básico de lectura la armas; este libro te va enseñando el vocabulario técnico poco a poco.

¿Ruby sigue siendo relevante? Sí. Rails sigue moviendo miles de negocios reales (tiendas en línea, plataformas, startups), Ruby sigue siendo top en scripting y automatización, y la demanda de programadores Ruby es estable porque hay menos competencia que en otros lenguajes. Además, lo que aprendas aquí (lógica, objetos, buenas prácticas) te sirve en cualquier lenguaje.

¿Cuánto tiempo me va a tomar terminar el libro? Depende de tu ritmo, pero con 30 a 60 minutos diarios, la mayoría termina en 6 a 10 semanas. No hay prisa: es mejor tardar 10 semanas entendiéndolo todo que 3 semanas memorizando a medias.

¿Puedo estudiar solo con el celular? Para leer, sí; para programar, no te lo recomiendo. Necesitas una computadora para instalar Ruby, usar irb y escribir programas de verdad. El celular sirve para repasar la teoría en el camión, pero la práctica se hace en la compu.

¿Qué hago si me atoro en un tema? Primero, relee la unidad con calma y corre los ejemplos a mano. Segundo, experimenta en irb con variaciones del ejemplo. Tercero, busca el mensaje de error o el concepto en internet (en español o inglés). Cuarto, descansa: a veces la solución llega después de dormir. Lo que no debes hacer es saltarte el tema y seguir como si nada.

¿El examen final es obligatorio? No, pero te lo recomiendo mucho. Es la forma honesta de medir si de verdad dominas el contenido. Si sacas menos de 35, repasa las unidades donde fallaste y vuelve a intentarlo. Nadie te está calificando más que tú mismo.

Tu entorno de trabajo: déjalo listo desde hoy

Un consejo práctico: dedica tu primera sesión a dejar tu entorno listo, para que después solo te concentres en programar.

Instala Ruby siguiendo los pasos de la Unidad 1 para tu sistema operativo. Verifica que quedó bien abriendo una terminal y escribiendo ``ruby -v``: si te muestra un número de versión, ya estás del otro lado.

Instala un editor de código. Visual Studio Code es gratis, funciona en todos los sistemas y tiene extensiones para Ruby que colorean el código y te avisan de errores. No necesitas configurarlo perfecto desde el día uno; con que puedas abrir archivos `.rb` y guardarlos, basta.

Crea una carpeta para tus prácticas, por ejemplo ``ruby-practicas``, y dentro de ella una subcarpeta por unidad: ``unidad-01``, ``unidad-02``, etc. Guarda ahí cada ejemplo que escribas y cada ejercicio que resuelvas. En unas semanas esa carpeta va a ser tu portafolio de aprendizaje y te va a servir para repasar.

Aprende desde hoy tres comandos de terminal: cómo moverte entre carpetas, cómo ver en qué carpeta estás y cómo correr un archivo Ruby con ``ruby mi_programa.rb``. No necesitas ser experto en la terminal, pero estos tres comandos los vas a usar todos los días.

Y por último, abre `irb` y escribe tu primera línea: ``puts "Hola, Ruby"``. Si ves el saludo en la pantalla, felicidades: tu entorno funciona y ya escribiste Ruby. Lo demás es seguir construyendo sobre eso.

UNIDAD 1: Bienvenido a Ruby

Bienvenido a la primera unidad de tu curso completo de Ruby. Si vienes de otros lenguajes como JavaScript, Python o Java, te vas a llevar una sorpresa agradable: Ruby es el lenguaje que fue diseñado para que el programador sea feliz. Eso no es un eslogan de marketing, es literalmente la filosofía que su creador, Yukihiro "Matz" Matsumoto, puso como norte del proyecto desde 1993. En esta unidad vas a entender por que Ruby se siente distinto, vas a instalarlo en tu computadora sin importar si usas Windows, Mac o Linux, vas a escribir tu primer programa "Hola, mundo" y vas a dominar los tres comandos mas basicos para mostrar informacion en pantalla: puts, print y p. También vas a aprender la sintaxis elegante que hace famoso a Ruby y como dejar comentarios que tus futuros colegas (y tu yo del futuro) van a agradecer. Esta unidad es la base de todo lo que sigue: sin una instalacion correcta y sin entender la forma de pensar de Ruby, el resto del curso seria como construir una casa sin cimientos. Al terminar estas tres clases vas a tener Ruby corriendo en tu maquina, vas a saber escribir codigo que se lee casi como espanol y vas a entender por que tantos programadores dicen que después de probar Ruby, escribir codigo en otros lenguajes se siente como hacer papeleo.

Explicacion detallada

Que es Ruby y por que importa

Ruby es un lenguaje de programacion de proposito general, interpretado, dinamico y orientado a objetos. Pero dejame traducir esa jerga a lenguaje humano. "Proposito general" significa que sirve para casi todo: paginas web, scripts de automatizacion, herramientas de linea de comandos, videojuegos y hasta inteligencia artificial. "Interpretado" significa que no necesitas compilar tu codigo antes de correrlo: escribes y ejecutas, directo. "Dinamico" significa que no tienes que declarar que tipo de dato es cada variable; Ruby lo descubre solo. Y "orientado a objetos" significa que TODO en Ruby es un objeto: los numeros, los textos, hasta el valor nulo (nil) son objetos con sus propios metodos. Esto ultimo es importantisimo y lo vamos a repetir mucho: en Ruby, todo es un objeto.

Para que te hagas una idea con una analogia: muchos lenguajes son como herramientas de un taller industrial, potentes pero con manuales de 500 paginas y procedimientos estrictos. Ruby es como una navaja suiza de alta gama: compacta, elegante y lista para usarse al instante. La diferencia no es solo estetica, es filosofica. En lenguajes como Java tienes que escribir ceremonias completas para hacer cosas simples (una clase, un metodo main, tipos declarados por todos lados) solo para imprimir un saludo. En Ruby escribes una linea y listo.

Hablando de industrias: Ruby es el lenguaje detras de Ruby on Rails, el framework con el que se construyeron empresas como GitHub, Shopify, Airbnb (en sus inicios), Basecamp y Twitch. Cuando escuchas que una startup levanto millones y lanzo su producto en semanas, muchas veces hay Rails detras. Pero Ruby no vive solo de Rails: es el lenguaje favorito para scripts de DevOps, para herramientas como Vagrant y Homebrew (el instalador de paquetes mas popular de Mac esta escrito en Ruby), y para automatizar tareas aburridas. Aprender Ruby te abre dos puertas al mismo tiempo: la de la programacion general y la del desarrollo web con Rails, que veremos en la unidad 10 de este curso.

Un error comun de principiante es pensar que Ruby es un lenguaje "de juguete" porque su sintaxis es amable. Nada mas lejos de la realidad: Ruby impulsa sistemas que procesan millones de dolares al dia (Shopify procesa ventas masivas en Ruby). La amabilidad de la sintaxis es una decision de diseno, no una limitacion. Es como pensar que un coche automatico es menos coche que uno manual: la transmision no define la potencia del motor.

La historia: Matz y la busqueda de la felicidad del programador

La historia de Ruby empieza en 1993 en Japon. Yukihiro Matsumoto, conocido como Matz, era un programador que usaba Perl y Python y sentia que algo faltaba: Perl era potente pero su sintaxis parecia jeroglificos, y Python era limpio pero le parecia rigido. Matz queria un lenguaje que combinara lo mejor de varios mundos: la orientacion a objetos pura de Smalltalk, la practicidad de Perl, la simplicidad de Lisp y la legibilidad de lenguajes como Ada o Eiffel. El 24 de febrero de 1993 empezo a escribirlo, y en diciembre de 1995 libero la primera version publica.

El nombre "Ruby" (rubi) no es casualidad: en ingles, Perl significa "perla", y el rubi es la piedra preciosa que sigue a la perla en la escala de dureza de Mohs y también es la piedra de nacimiento de julio, mientras la perla es la de junio. Matz bromeo diciendo que queria una joya mas valiosa que la perla. Ese sentido del humor y esa calidez humana impregnan toda la comunidad Ruby.

La frase mas famosa de Matz, y la que debes grabarte, es esta: "Quería hacer un lenguaje que me hiciera feliz programando. A menudo la gente, especialmente los ingenieros en

computacion, se enfocan en las maquinas. Piensan: haciendo esto, la maquina correra mas rapido. Haciendo esto, la maquina correra mas eficientemente. Pero en realidad necesitamos enfocarnos en los humanos, en como los humanos se preocupan por hacer programacion o manejar la aplicacion de las maquinas. Nosotros somos los amos, ellas son las esclavas." Ese es el corazon de Ruby: el lenguaje esta disenado para TI, no para la maquina.

Ruby crecio despacio en Japon durante los 90, pero exploto mundialmente en 2004-2005 cuando David Heinemeier Hansson (DHH) libero Ruby on Rails y lo uso para construir Basecamp. De pronto, programadores de todo el mundo descubrieron que podian construir aplicaciones web completas en dias en lugar de meses. La comunidad Ruby se volvio famosa por su amabilidad, por sus conferencias llenas de energia y por su cultura de codigo elegante y bien probado.

Analogia para que no se te olvide: si los lenguajes de programacion fueran restaurantes, C seria un asador argentino donde tu mismo cortas la carne con tu cuchillo (control total, mas trabajo), Java seria un restaurante de lujo con codigo de vestimenta y reservacion (ceremonia y reglas), Python seria un restaurante minimalista nordico (limpio, ordenado, una sola forma de hacer las cosas) y Ruby seria la fonda de tu abuela donde la cocinera te conoce por tu nombre, te sirve generoso y te pregunta si quieres mas: calido, flexible y pensado para que disfrutes.

La filosofia de Ruby: principios que debes interiorizar

Ruby tiene varios principios de disenio que vas a ver reflejados en cada linea de codigo que escribas. Entenderlos ahora te va a ahorrar meses de confusion.

Principio 1: Todo es un objeto. En muchos lenguajes hay una division entre "tipos primitivos" (numeros, caracteres) y "objetos de verdad". En Ruby no existe esa division: el numero 5 es un objeto de la clase Integer, el texto "hola" es un objeto de la clase String, e incluso nil (la nada) es un objeto de la clase NilClass. Esto significa que puedes llamar metodos sobre cualquier cosa: 5.times, "hola".upcase, nil.nil?. Cuando interiorices esto, la documentacion de Ruby se vuelve predecible: todo lo que ves tiene metodos que puedes usar.

Principio 2: El principio de la menor sorpresa (Principle of Least Surprise, POLS). Matz disenio Ruby para que se comporte como un programador esperaria que se comporte. Si algo se siente natural, probablemente funciona asi. Por ejemplo, los arreglos empiezan en el indice 0 (como casi todos los lenguajes), los metodos que terminan en ? devuelven verdadero o falso (empty?, nil?), y los que terminan en ! son versiones "peligrosas" que modifican el objeto. Esta convencion de ? y ! no existe en casi ningun otro lenguaje popular, y es oro puro para leer codigo ajeno.

Principio 3: Hay mas de una forma de hacerlo (TIMTOWTDI, heredado de Perl). A diferencia de Python, que tiene una sola forma "correcta", Ruby te da libertad. Puedes escribir un if en una linea o en cinco, puedes omitir parentesis, puedes usar bloques de varias formas. Esta libertad es un arma de doble filo: te permite escribir codigo hermoso, pero también codigo confuso si abusas. La buena noticia es que la comunidad desarrollo convenciones claras (la guia de estilo de Ruby) que te dicen cual forma elegir en cada caso. Libertad con responsabilidad.

Principio 4: Los bloques y la expresividad. Ruby te permite pasar pedazos de codigo como si fueran datos (los bloques), lo que hace que operaciones comunes se lean como frases en ingles: `3.times { pon "hola" }`, `[1,2,3].each { |n| pon n * 2 }`. Esto hace que Ruby se lea casi como pseudocodigo ejecutable. Cuando un lenguaje se lee facil, los errores se encuentran facil y el codigo se mantiene facil.

Principio 5: Duck typing ("si camina como pato y hace cuac como pato, es un pato"). En Ruby no te importa de que clase es un objeto, te importa que metodos responde. Si un objeto responde a `each`, lo puedes iterar; si responde a `to_s`, lo puedes convertir a texto. Esto da una flexibilidad enorme y es la base de la filosofia dinamica del lenguaje.

Error comun de principiante: venir de Java o C# y tratar de programar en Ruby como si fuera Java con otra sintaxis: crear clases con getters y setters manuales larguissimos, declarar tipos en comentarios, evitar los bloques por "miedo" a la sintaxis nueva. Codigo malo (estilo Java en Ruby):

```
class Persona
  def initialize(nombre)
    @nombre = nombre
  end
  def get_nombre()
    return @nombre
  end
  def set_nombre(nuevo_nombre)
    @nombre = nuevo_nombre
  end
end
```

Codigo bueno (estilo Ruby):

```
class Persona
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end
```

El código bueno hace lo mismo en la mitad de líneas y usa las convenciones del lenguaje. Moraleja: no traduzcas tu lenguaje anterior a Ruby; piensa en Ruby. Es como mudarte a otro país: puedes sobrevivir hablando tu idioma con acento, pero vas a vivir mejor si aprendes el idioma local.

Buena práctica de la industria: las empresas que contratan programadores Ruby no solo evalúan si tu código funciona, evalúan si tu código "parece Ruby" (lo llaman código idiota). Leer la guía de estilo oficial de Ruby (rubystyle.guide) y usar herramientas como RuboCop (un "corrector" automático de estilo) es lo que separa a un programador que escribe Ruby de un programador que escribe en Ruby. Desde el día uno, acostúmbrate a los nombres en snake_case (palabras_separadas_por_guiones_bajos), a la indentación de dos espacios y a omitir los parentesis cuando no aportan claridad.

Instalación en Windows paso a paso

Vamos a instalar Ruby en tu computadora. Elige la sección de tu sistema operativo, pero te recomiendo leer las tres para entender las diferencias: en entrevistas de trabajo te pueden preguntar.

En Windows, la forma oficial y más sencilla es usar RubyInstaller, un proyecto que empaqueta Ruby con todo lo necesario para Windows. Paso 1: abre tu navegador y ve a rubyinstaller.org (es el sitio oficial; desconfía de cualquier otro sitio que te ofrezca "Ruby para Windows"). Paso 2: descarga la versión marcada como recomendada; verás opciones como "Ruby+Devkit". El Devkit incluye herramientas de compilación necesarias para instalar gemas (las librerías de Ruby) que traen código en C. Descarga la versión "with Devkit". Paso 3: ejecuta el instalador. En las opciones, marca las tres casillas importantes: "Add Ruby executables to your PATH" (esto te permite correr ruby desde cualquier terminal), "Associate .rb and .rbw files" (para que los archivos Ruby se abran correctamente) y deja marcada la instalación del MSYS2/devkit al final. Paso 4: cuando termine la instalación principal, se abrirá una terminal negra que te pregunta qué componentes instalar; escribe 1, 2 y 3 (o simplemente presiona Enter para la opción por defecto) y espera a que termine. Puede tardar varios minutos, es normal.

Para verificar que todo quedó bien, abre una terminal nueva (cmd o PowerShell) y escribe:

```
ruby -v
```

Deberias ver algo como ruby 3.3.4 (2024-07-03 revision ...) [x64-mingw-ucrt]. Si ves el numero de version, felicidades: Ruby esta instalado. También verifica el manejador de gemas:

```
gem -v
```

Veras un numero como 3.5.16. Si alguno de los dos comandos dice "no se reconoce como comando", lo mas probable es que la terminal se abrio antes de terminar la instalacion o que no se marco la casilla del PATH; cierra la terminal, abre una nueva y prueba de nuevo. Si sigue fallando, reinstala marcando la casilla del PATH.

Error comun en Windows: instalar Ruby desde la Microsoft Store o desde instaladores no oficiales y luego no poder instalar gemas porque falta el Devkit.Codigo malo (lo que hace el principiante frustrado): descargar un ZIP suelto de Ruby, descomprimirlo en el escritorio y tratar de correr ruby.exe a mano. Codigo bueno: usar RubyInstaller con Devkit y verificar con ruby -v y gem -v antes de seguir. Otro error tipico: tener dos versiones de Ruby instaladas (una de la Store y otra de RubyInstaller) que se pelean por el PATH. Solucion: desinstala la de la Store desde Configuracion > Aplicaciones y deja solo RubyInstaller.

Instalacion en Mac paso a paso

En Mac tienes dos caminos: el facil y el profesional. El facil es instalarlo con Homebrew. El profesional también usa Homebrew, pero para instalar un manejador de versiones. Te explico ambos porque en el mundo real vas a necesitar el profesional tarde o temprano.

Primero verifica si ya tienes Ruby: Mac trae un Ruby del sistema preinstalado, pero es viejo y Apple lo pone ahi para sus propias herramientas, no para ti. NO uses el Ruby del sistema para programar: si lo rompes, rompes herramientas del sistema. Compruebalo:

```
ruby -v
```

Si dice algo como ruby 2.6.x, ese es el del sistema. Ignoralo.

Camino profesional (recomendado): instala Homebrew si no lo tienes. Abre la Terminal (la encuentras en Aplicaciones > Utilidades) y ejecuta el comando de instalacion desde brew.sh. Una vez instalado Homebrew, instala rbenv, el manejador de versiones mas popular:

```
brew install rbenv ruby-build
```

Luego agrega rbenv a tu shell (sigue las instrucciones que te muestra el comando `rbenv init`) e instala la version de Ruby que quieras:

```
rbenv install 3.3.4
rbenv global 3.3.4
```

Cierra y abre la Terminal, y verifica con `ruby -v`: ahora deberia decir 3.3.4 y which ruby deberia apuntar a tu carpeta de usuario (`~/.rbenv`), no a `/usr/bin/ruby`. La ventaja de rbenv es que puedes tener varias versiones de Ruby instaladas y cambiar entre ellas por proyecto, algo indispensable cuando trabajas en varios proyectos con requisitos distintos.

Camino facil: `brew install ruby`. Esto instala la ultima version estable y la pone en tu PATH. Funciona, pero cuando necesites otra version vas a desear haber usado rbenv. Mi recomendacion de colega: invierte los 10 minutos extra en rbenv desde el principio.

Error comun en Mac: instalar gemas con `sudo` (`sudo gem install rails`). Esto instala cosas como administrador y mezcla tu Ruby con el del sistema; tarde o temprano algo se rompe y los errores son cripticos.Codigo malo: `sudo gem install <gema>`.Codigo bueno: `gem install <gema>` usando tu Ruby de rbenv, sin `sudo`, porque rbenv instala todo en tu carpeta de usuario donde tienes permiso total. Si algun tutorial te pide `sudo` para instalar una gema, ese tutorial esta desactualizado o asume el Ruby del sistema: no lo sigas.

Instalacion en Linux paso a paso

En Linux también tienes el camino facil (el gestor de paquetes de tu distribucion) y el camino profesional (rbenv o rvm). El gestor de paquetes suele traer versiones viejas de Ruby, asi que el camino profesional es el estandar en la industria.

Para Ubuntu/Debian, primero instala las dependencias de compilacion (Ruby se compila desde el codigo fuente cuando usas rbenv):

```
sudo apt update
sudo apt install -y build-essential libssl-dev libreadline-dev zlib1g-dev
libyaml-dev libffi-dev
```

Luego instala rbenv siguiendo su documentacion oficial (es un git clone a `~/.rbenv` mas la configuracion de tu shell), instala ruby-build como plugin, y luego:

```
rbenv install 3.3.4
rbenv global 3.3.4
```

En Fedora/CentOS los paquetes se llaman distinto (gcc, openssl-devel, readline-devel, zlib-devel, libyaml-devel, libffi-devel) pero la idea es identica: dependencias, rbenv, instalar, verificar. En Arch Linux, el paquete ruby de pacman suele estar razonablemente actualizado y muchos desarrolladores lo usan directo, aunque rbenv sigue siendo la opcion mas flexible.

Verificacion final en cualquier sistema: estos tres comandos deben funcionar:

```
ruby -v
gem -v
irb
```

El tercero, irb, es la consola interactiva de Ruby (Interactive Ruby): te abre un prompt donde puedes escribir codigo Ruby y ver el resultado al instante. Escribe `2 + 2`, presiona Enter y veras 4. Para salir, escribe `exit`. irb va a ser tu mejor amigo en este curso: cada vez que tengas duda de como funciona algo, prueba en irb antes de escribir un archivo. Los programadores Ruby profesionales viven en irb (o en su primo moderno, pry).

Analogia: instalar Ruby con el manejador de versiones es como tener un closet organizado con ganchos etiquetados (rbenv) en lugar de aventar toda la ropa en una pila (instalaciones mezcladas). Al principio la pila parece mas rapida; a los tres meses no encuentras nada.

Tu primer programa: Hola, mundo

Toda la tradicion indica que el primer programa debe saludar al mundo, y no vamos a romper la tradicion. Abre tu editor de texto favorito (VS Code es la eleccion mas popular; Sublime Text y RubyMine también son excelentes) y crea un archivo llamado `hola.rb`. La extension `.rb` le dice al sistema que es un archivo Ruby. Escribe esto:

```
puts "Hola, mundo"
```

Guardalo y, en tu terminal, navega a la carpeta donde lo guardaste y ejecuta:

```
ruby hola.rb
```

Veras impreso `Hola, mundo`. Eso es todo. Sin clases, sin metodo main, sin punto y coma, sin compilacion. Una linea que hace exactamente lo que dice. Detente un momento a apreciar esto si vienes de Java, donde el mismo programa requiere como cinco lineas de ceremonia.

Que esta pasando tecnicamente: ruby es el interprete; lee tu archivo de arriba a abajo y ejecuta cada linea. `puts` es un metodo (una accion) que imprime texto en la pantalla y agrega

un salto de linea al final. "Hola, mundo" es un string, un objeto de texto. Cuando el interprete ve puts "Hola, mundo", le dice al objeto string: "muestrate en pantalla". Asi de directo.

Buena practica desde el dia uno: guarda tus archivos de practica en una carpeta organizada, por ejemplo ~/ruby-practicas/unidad1/, con nombres descriptivos en minusculas y guiones bajos: hola_mundo.rb, mi_edad.rb, calculadora.rb. Evita espacios en los nombres de archivo y evita la carpeta del escritorio llena de archivos sueltos llamados prueba1.rb, prueba2_final.rb, prueba2_final_DE_VERDAD.rb. Tu yo del futuro te lo va a agradecer.

Error comun: guardar el archivo como hola.rb.txt porque el Bloc de notas de Windows agrega la extension oculta. Si ruby hola.rb dice que no encuentra el archivo, revisa en el explorador que la extension real sea .rb (activa "ver extensiones de archivo" en las opciones de carpeta). Otro error: ejecutar ruby hola.rb desde una carpeta distinta a donde guardaste el archivo. La terminal tiene una "ubicacion actual"; usa cd para moverte a tu carpeta de practicas antes de ejecutar.

La sintaxis elegante de Ruby: menos ruido, mas significado

La sintaxis es la forma en que escribes el codigo, y la de Ruby fue disenada para que el codigo se lea como una explicacion, no como un acertijo. Vamos a ver las decisiones de diseno que la hacen elegante.

Primero: no necesitas punto y coma al final de las lineas. En Ruby el salto de linea termina la instruccion. Puedes poner punto y coma si quieres separar dos instrucciones en la misma linea (puts "hola"; puts "adios"), pero es raro y la guia de estilo lo desaconseja. Menos simbolos = menos ruido visual.

Segundo: la indentacion es de dos espacios, no de cuatro y no con tabuladores. Esto no es obligatorio para el interprete (Ruby no es Python, no te obliga), pero es la convencion universal de la comunidad. Si escribes con cuatro espacios, tu codigo funciona igual, pero cualquier programador Ruby que lo vea va a saber al instante que eres nuevo. Configura tu editor para que el tabulador inserte dos espacios en archivos .rb.

Tercero: los parentesis en las llamadas a metodos son opcionales cuando no hay ambigüedad. Puedes escribir puts("Hola") o puts "Hola"; ambas funcionan. La convencion: omite los parentesis en comandos simples (puts "hola", attr_reader :nombre) y usalos cuando el codigo se vuelve dificil de leer o cuando hay varios argumentos complejos. Esta flexibilidad es parte de lo que hace que Ruby se lea como lenguaje humano: puts "Hola, mundo" se lee como una orden directa, mientras puts("Hola, mundo") se lee como una llamada tecnica. Ambas validas, elige según claridad.

Cuarto: el return es implicito. En Ruby, un metodo devuelve automaticamente el valor de la ultima expresion evaluada, sin necesidad de escribir return. Puedes escribir return si quieres salir antes de tiempo, pero en el caso normal se omite. Esto elimina una palabra reservada de cada metodo y hace el codigo mas limpio. Ejemplo:

```
def cuadrado(n)
  n * n
end
```

Ese metodo devuelve $n * n$ sin decir return. Si vienes de otros lenguajes esto se siente raro al principio, como si faltara algo; con el tiempo se siente liberador, como quitarte los zapatos al llegar a casa.

Quinto: las convenciones de nombres lo dicen todo. En Ruby los nombres no son solo etiquetas, son documentacion: las variables y metodos usan snake_case (mi_variable, calcular_total), las clases y modulos usan CamelCase (MiClase, MiModulo), las constantes usan MAYUSCULAS_CON_GUIONES (PI, MAX_INTENTOS), los metodos que devuelven verdadero/falso terminan en ? (vacio?, incluido?), y los metodos "peligrosos" que modifican el objeto original terminan en ! (ordenar! vs ordenar). Cuando lees codigo Ruby ajeno, estas convenciones te permiten adivinar que hace un metodo antes de leer su implementacion. Es como el codigo de colores de los cables electricos: nadie te obliga a usarlos, pero ignorarlos es pedir problemas.

Sexto: todo es una expresion. En Ruby casi todo devuelve un valor: un if devuelve un valor, un case devuelve un valor, incluso una asignacion devuelve un valor. Esto permite escribir codigo muy expresivo:

```
mensaje = if hora < 12 then "Buenos dias" else "Buenas tardes" end
```

En otros lenguajes el if es una "sentencia" que no devuelve nada y necesitas escribir mas lineas. En Ruby el if es una expresion que produce un valor que puedes guardar. Esta idea, heredada de Lisp, es una de las razones por las que el codigo Ruby tiende a ser mas corto.

Error comun de principiante: abusar de la flexibilidad y escribir codigo "inteligente" pero ilegible, como encadenar cinco metodos sin parentesis ni espacios claros, u omitir parentesis donde si hacian falta y generar errores extranos. Codigo malo:

```
puts [1, 2, 3].map { |n| n * 2 }.select { |n| n > 2 }.join ", "
```

No es que este mal sintacticamente, pero mezclar estilos confunde. Código bueno: se consistente, usa parentesis cuando la expresion es compleja y divide en lineas cuando la cadena es larga. La elegancia no es escribir lo menos posible, es escribir lo mas claro posible. Un programador senior me dijo una vez: "escribe codigo como si la persona que lo va a mantener fuera un psicopata violento que sabe donde vives". En Ruby, la elegancia y la claridad son lo mismo.

Buena practica de la industria: instala RuboCop desde el principio (gem install rubocop) y correlo sobre tus archivos de practica. Te va a marcar desviaciones de estilo con explicaciones. Al principio es molesto, como un maestro estricto; a los dos meses tu codigo sale limpio por instinto y en entrevistas tecnicas eso se nota.

Comentarios: el arte de explicar el por que

Los comentarios son notas que le dejas al lector humano del codigo; el interprete los ignora por completo. En Ruby hay dos formas: el comentario de una linea con # y el comentario de varias lineas con =begin y =end.

```
# Esto es un comentario de una linea: Ruby lo ignora
puts "Hola" # Los comentarios también pueden ir al final de una linea

=begin
Esto es un comentario
de varias lineas.
Casi nadie lo usa en codigo real.
=end
puts "Adios"
```

El comentario multilinea con =begin/=end existe pero en la practica casi no se usa; la comunidad prefiere varias lineas con #. Lo menciono para que lo reconozcas si lo ves en codigo viejo, no para que lo uses.

Ahora, la parte importante no es COMO comentar, sino QUE comentar. Regla de oro: los comentarios explican el POR QUE, no el QUE. El codigo ya dice que hace; el comentario debe decir por que se hace asi, que decision de negocio hay detras, o que trampa esconde esa linea.

Código malo (comentarios que repiten lo obvio):

```
# Asigna 5 a la variable x
x = 5
# Imprime x en pantalla
puts x
# Suma 1 a x
x = x + 1
```

Estos comentarios son ruido: no aportan nada que el código no diga. Peor aun, se desactualizan: alguien cambia el código y olvida el comentario, y ahora el comentario miente. Un comentario mentiroso es peor que ningún comentario.

Código bueno (comentarios que explican decisiones):

```
# El proveedor cobra en centavos, por eso multiplicamos por 100
# antes de enviar el monto a su API (documentacion v2.3, seccion pagos).
total_centavos = total * 100
```

Ese comentario vale oro: explica una decisión que el código por sí solo no revela y cita la fuente. Otro buen uso: advertir sobre trampas.

```
# OJO: el metodo sort! modifica el arreglo original; usar sort
# si necesitas conservar el orden de llegada para el reporte.
precios.sort!
```

Buena práctica de la industria: en proyectos reales, los comentarios buenos son escasos y quirúrgicos. La meta es escribir código tan claro que casi no necesite comentarios, y reservarlos para el por qué. Además, Ruby tiene una convención especial: las líneas que empiezan con `#!` (shebang) o con `# frozen_string_literal: true` tienen significado para el intérprete o las herramientas, no son comentarios decorativos. Y la documentación formal se escribe con RDoc o YARD usando comentarios con formato especial encima de clases y métodos, algo que veremos en unidades avanzadas.

Analogía: los comentarios son como las notas al margen en un libro de cocina. Nadie necesita que le escribas "aquí se bate el huevo" al lado de la instrucción "bata el huevo". Pero una nota que diga "mi abuela usaba mantequilla en vez de aceite porque el horno de la casa calienta de más" vale más que la receta misma.

puts, print y p: los tres mosqueteros de la salida

Ruby te da tres formas principales de mostrar informacion en pantalla, y cada una tiene su personalidad. Dominarlas es esencial porque las vas a usar para depurar (ver que esta pasando dentro de tu programa) durante todo el curso.

puts (put string) imprime el texto y agrega un salto de linea al final. Es el que mas vas a usar. Si le pasas un arreglo, imprime cada elemento en su propia linea (detalle util que sorprende a los nuevos).

```
puts "Hola"
puts "Mundo"
# Imprime:
# Hola
# Mundo
# (cada uno en su linea)
```

print hace lo mismo pero SIN el salto de linea final. Sirve cuando quieres construir una linea por partes o cuando pides datos al usuario y quieres que escriba en la misma linea del mensaje.

```
print "Como te llamas? "
# El cursor se queda en la misma linea, esperando tu respuesta
```

p es el inspector: imprime la representacion tecnica del objeto usando su metodo inspect. Muestra las comillas de los strings, la estructura de los arreglos y detalles que puts oculta. Es tu herramienta de depuracion rapida.

```
nombre = "Ana"
puts nombre # imprime: Ana
p nombre    # imprime: "Ana" (con comillas, ves que es un string)
```

La diferencia parece sutil pero es crucial cuando depuras: si una variable trae un espacio invisible al final ("Ana "), puts te mostrara "Ana" y no veras el problema, pero p te mostrara "Ana " con el espacio a la vista. Los programadores Ruby usan p (o su primo pp, pretty print) para inspeccionar datos, y puts para mostrar resultados al usuario.

Comparacion directa con codigo:

```
frutas = ["manzana", "pera"]
puts frutas
# manzana
# pera
print frutas
# ["manzana", "pera"] (sin salto de linea al final)
p frutas
# ["manzana", "pera"] (con salto de linea, formato de inspeccion)
```

Error comun de principiante: usar puts para depurar y no ver el problema porque puts "maquilla" la salida.Codigo malo: tienes un bug donde un string trae un salto de linea escondido, imprimes con puts, todo "se ve bien" y pierdes una hora. Codigo bueno: ante la duda, depura con p, que te muestra la verdad cruda del objeto. Otro error: olvidar que puts devuelve nil (no devuelve lo que imprimio), asi que poner puts dentro de una expresion esperando usar su valor te da sorpresas. Ejemplo tramposo:

```
resultado = puts "hola"
p resultado # nil, no "hola"
```

Buena practica: en codigo real, puts/print son para scripts y herramientas de consola; en aplicaciones web (Rails) se usan registros de log (logger) en vez de puts, porque en un servidor nadie ve la pantalla. Pero para aprender y para scripts, puts es perfecto.

También existe printf para formato avanzado (heredado de C), util cuando quieres columnas alineadas o decimales controlados:

```
printf("Nombre: %s, Edad: %d\n", "Luis", 30)
# Nombre: Luis, Edad: 30
```

No te preocupes si printf se ve criptico ahora; lo retomaremos cuando hablemos de strings avanzados. Por ahora quedate con: puts para mostrar, print para pedir datos en la misma linea, p para inspeccionar la verdad.

El flujo de trabajo del programador Ruby: edita, ejecuta, repite

Antes de los ejemplos, dejame describirte el ciclo de trabajo que vas a repetir miles de veces: escribes codigo en tu editor, lo guardas, lo ejecutas con ruby archivo.rb en la terminal, lees el resultado o el error, corriges y repites. Suena trivial, pero hacerlo con disciplina marca la diferencia entre avanzar y atorarse.

Cuando Ruby encuentra un error, te muestra un mensaje con el numero de linea y el tipo de error. Al principio estos mensajes parecen estar en otro idioma; con practica se vuelven tu mapa del tesoro. Ejemplo: si escribes `puts "hola (sin cerrar la comilla)`, veras algo como "unterminated string meets end of file". Traduccion: "empezaste un string y nunca lo cerraste". El numero de linea te dice donde buscar.

Regla de oro para principiantes: cuando algo falle, antes de cambiar codigo al azar, lee el mensaje de error completo y prueba la parte sospechosa en irb. El 80% de los atorones se resuelven asi. El otro 20% se resuelve explicandole el problema a alguien (o a un pato de hule en tu escritorio: la tecnica del "rubber duck debugging" es real y funciona porque te obliga a ordenar tus ideas).

Buena practica: guarda versiones de tus programas con frecuencia. No necesitas Git todavia (lo veras mas adelante), pero al menos duplica el archivo antes de un cambio arriesgado: `calculadora.rb` y `calculadora_respaldo.rb`. Perder una hora de trabajo por no tener respaldo es un rito de iniciacion que puedes saltarte si me haces caso.

Ejemplos paso a paso

Ejemplo 1: tu primer programa comentado. Crea el archivo `mi_primer.rb` con este contenido. Cada linea tiene un proposito: el comentario inicial describe el programa, `puts` muestra mensajes, y el comentario final marca el cierre. Ejecuta con `ruby mi_primer.rb`.

```
# Mi primer programa en Ruby
# Autor: yo, futuro programador Ruby
# Fecha: hoy, el día que empece

puts "Hola, mundo"
puts "Estoy aprendiendo Ruby en Lucas EGA Academy"

# Fin del programa
```

Al ejecutarlo veras las dos lineas impresas y nada de los comentarios: el interprete los salto. Fijate en que no hubo que importar nada ni declarar nada; Ruby ejecuta de arriba a abajo.

Ejemplo 2: la diferencia entre `puts`, `print` y `p` en accion. Guarda como `salidas.rb` y ejecutalo para VER la diferencia con tus propios ojos, no solo leerla.

```
puts "=== Con puts ==="
puts "manzana"
puts "pera"

puts "=== Con print ==="
print "manzana"
print "pera"
puts "" # salto de linea para ordenar la salida

puts "=== Con p ==="
p "manzana"
p ["manzana", "pera"]
```

La salida muestra: puts separa en lineas, print pega todo (manzanapera), y p revela la estructura con comillas y corchetes. Este programita es un laboratorio: modificalo, agrega tus propios valores y observa.

Ejemplo 3: pedir datos con print y leer con gets (adelanto de la unidad 2, pero util desde ya). Guarda como saludo.rb. print deja el cursor en la misma linea para que la respuesta se vea ordenada.

```
print "Como te llamas? "
nombre = gets.chomp
puts "Mucho gusto, " + nombre
```

gets lee lo que escribas y chomp le quita el salto de linea final. La ultima linea concatena textos con +. Pruebalo escribiendo tu nombre. En la unidad 2 diseccionaremos gets.chomp a fondo; por ahora usalo como receta.

Ejemplo 4: comentarios que si aportan. Guarda como receta.rb y nota como los comentarios explican decisiones, no repiten codigo.

```
# Calculadora de propina para el restaurante de mi tio.
# En Mexico la propina usual es 10-15%; usamos 15% porque
# el servicio aqui incluye meseros dedicados por mesa.

cuenta = 850
propina = cuenta * 0.15 # 15% en decimal

puts "Cuenta: $850"
puts "Propina (15%): "
p propina # p para verificar el numero exacto con decimales
puts "Total a pagar: "
p cuenta + propina
```

Observa el uso de p para inspeccionar numeros con decimales exactos: puts redondearia la presentacion en algunos casos, p te muestra el valor real. Los comentarios cuentan la historia del negocio detras del numero 0.15.

Ejemplo 5: sintaxis elegante en accion, comparando estilos. Guarda como estilos.rb. Muestra que Ruby permite varias formas y cual prefiere la comunidad.

```
# Estilo con parentesis (valido, pero mas ruidoso)
puts("Hola con parentesis")

# Estilo idiomatico Ruby (preferido para comandos simples)
puts "Hola sin parentesis"

# Return implicito: el metodo devuelve la ultima expresion
def doble(n)
  n * 2
end

p doble(21) # 42, sin haber escrito return
```

Ejecuta y confirma que ambas formas de puts funcionan y que doble devuelve 42. El metodo doble demuestra el return implicito: la ultima linea es el resultado.

Ejemplo 6: todo es un objeto, demostracion practica. Guarda como objetos.rb. Este ejemplo te va a volar la cabeza si vienes de lenguajes con tipos primitivos.


```
p 5.class      # Integer: el numero 5 es un objeto
p "hola".class # String: el texto es un objeto
p nil.class    # NilClass: hasta la nada es un objeto

# Y como son objetos, tienen metodos:
p 5.even?      # false (5 no es par)
p "hola".upcase # "HOLA"
p nil.nil?     # true
```

Cada linea llama un metodo sobre un "dato simple". En Ruby no hay ciudadanos de segunda clase: todo responde a metodos. Guarda esta idea, es la llave maestra del lenguaje.

Ejemplo 7: multiples instrucciones y el punto y coma (para reconocerlo, no para usarlo). Guarda como `puntoycoma.rb`.

```
puts "uno"; puts "dos"; puts "tres"
```

Funciona e imprime tres lineas, pero la guia de estilo dice que cada instruccion va en su linea. Lo incluyo para que cuando lo veas en codigo ajeno no te asustes, y para que entiendas por que no debes escribir asi tu codigo nuevo.

Ejemplo 8: tu entorno verificado por codigo. Guarda como `diagnostico.rb`. Este programa comprueba tu instalacion e imprime informacion util; es el tipo de script que un profesional escribe el primer dia en una maquina nueva.

```
puts "=== Diagnostico de mi instalacion Ruby ==="
print "Version de Ruby: "
p RUBY_VERSION # constante con la version actual
print "Plataforma: "
p RUBY_PLATFORM # sistema operativo detectado
puts "Si ves estos datos, tu instalacion funciona."
puts "Sigue asi, vas bien."
```

`RUBY_VERSION` y `RUBY_PLATFORM` son constantes que Ruby define solo. Ejecutalo: si imprime tu version y plataforma, tu entorno esta listo para todo el curso. Si falla, regresa a la seccion de instalacion de tu sistema operativo.

Ejemplo 9: mini programa completo que junta todo lo aprendido. Guarda como `presentacion.rb`. Pide datos, usa comentarios utiles y muestra con los tres comandos.

```

# Programa de presentacion personal
# Pide nombre y edad, calcula el anio de nacimiento aproximado.

print "Tu nombre: "
nombre = gets.chomp
print "Tu edad: "
edad = gets.chomp.to_i # to_i convierte el texto a numero entero

anio_actual = 2026
nacimiento = anio_actual - edad

puts "=== Tu ficha ==="
puts "Nombre: " + nombre
p edad # inspeccionamos que la conversion funcione
puts "Naciste alrededor de: "
p nacimiento
puts "Bienvenido al curso de Ruby, " + nombre

```

Este programa usa print para pedir datos en la misma linea, gets.chomp para leer, to_i para convertir (lo veremos a fondo en la unidad 2) y mezcla puts con p según convenga. Es un programa "real" en miniatura: entrada, procesamiento y salida. Modificalo para que también pregunte tu ciudad y la incluya en la despedida.

Practica

Ejercicio 1: instala Ruby y verifica. Sigue los pasos de tu sistema operativo (Windows, Mac o Linux) e instala Ruby. Luego abre una terminal y ejecuta ruby -v y gem -v. Solucion explicada: no hay codigo que escribir aqui, el entregable son las dos versiones impresas en tu terminal. ruby -v debe mostrar algo como ruby 3.3.4 y gem -v un numero como 3.5.16. Si ves "comando no reconocido", el problema casi siempre es que la terminal se abrio antes de terminar la instalacion (cierrala y abre una nueva) o que no se agrego Ruby al PATH (en Windows, reinstala marcando la casilla). Este ejercicio parece trivial pero es el mas importante de la unidad: sin un entorno funcional, nada mas avanza. Tomate el tiempo de dejarlo perfecto.

Ejercicio 2: tu primer Hola, mundo personalizado. Crea hola_mundo.rb que imprima tres lineas: tu nombre, lo que esperas aprender de Ruby y el nombre de la academia (lucasegaacademy.com). Solucion explicada:

```
puts "Me llamo Carlos"
puts "Quiero aprender Ruby para crear paginas web con Rails"
puts "Estudio en lucasegaacademy.com"
```

Funciona porque puts imprime cada string en su propia linea con salto de linea automatico. Ejecutalo con ruby hola_mundo.rb. La gracia del ejercicio es mecanizar el ciclo edita-guarda-ejecuta, que vas a repetir miles de veces.

Ejercicio 3: laboratorio de puts vs print vs p. Escribe un programa que muestre la diferencia entre los tres usando tu comida favorita. Solucion explicada:

```
comida = "tacos"
puts "Con puts:"
puts comida
puts "Con print:"
print comida
print comida
puts ""
puts "Con p:"
p comida
```

La salida muestra: puts pone cada taco en su linea; print los pega (tacostacos) porque no agrega salto de linea; p muestra "tacos" con comillas revelando que es un string. El puts "" vacio después de los print es un truco comun para ordenar la salida: imprime solo el salto de linea.

Ejercicio 4: comentarios utiles. Toma el programa del ejercicio 2 y agregale un comentario inicial que describa el programa, tu nombre como autor y la fecha, mas un comentario al final de una linea explicando POR QUE elegiste esa meta (no que hace la linea). Solucion explicada:

```
# Programa de presentacion - Unidad 1
# Autor: Carlos - 2026-09-19
# Elegi aprender Ruby porque quiero construir mi propia
# aplicacion web y Rails es la ruta mas rapida para lograrlo.

puts "Me llamo Carlos" # mi nombre real, sin apodos
puts "Quiero aprender Ruby para crear paginas web con Rails"
puts "Estudio en lucasegaacademy.com"
```

Los comentarios buenos cuentan historia y motivacion; el comentario al final de la linea aclara una decision (usar el nombre real). Nota que NO comentamos "esta linea imprime mi nombre": eso seria ruido.

Ejercicio 5: metodos con signo de interrogacion. En irb, prueba estos metodos y anota que devuelven: 10.even?, 7.odd?, "".empty?, "hola".empty?, nil.nil?. Solucion explicada: 10.even? devuelve true (10 es par), 7.odd? devuelve true (7 es impar), "".empty? devuelve true (string vacio), "hola".empty? devuelve false, nil.nil? devuelve true. La leccion: en Ruby los metodos que terminan en ? siempre responden true o false, y puedes llamarlos sobre cualquier objeto porque todo es un objeto. Acostumbrate a leer el ? como la pregunta "es...?" y el codigo se vuelve autodocumentado.

Ejercicio 6: return implicito. Escribe un metodo saludo que reciba un nombre y devuelva el saludo completo SIN usar la palabra return, y compruebalo con p. Solucion explicada:

```
def saludo(nombre)
  "Hola, " + nombre + ", bienvenido a Ruby"
end

p saludo("Ana")
```

Imprime "Hola, Ana, bienvenido a Ruby". Funciona porque la ultima (y unica) expresion del metodo, la concatenacion de strings, se convierte automaticamente en el valor de retorno. Si hubieras escrito return seria valido pero no idiomatico; en Ruby se omite por convencion y los programadores experimentados lo esperan asi.

Ejercicio 7: diagnostico de errores. Este programa tiene dos errores; encuéntralos y corrígelos. Codigo con errores:

```
puts "Empezando..."
p "terminado"
```

Solucion explicada: el primer error es el string sin cerrar en puts "Empezando...": falta la comilla de cierre, y Ruby reportara "unterminated string meets end of file". El segundo error es mas sutil: como el string nunca se cerro, el interprete sigue leyendo y todo el resto del archivo se vuelve parte del string, asi que el error real enmascara la linea de p. Codigo corregido:

```
puts "Empezando..."
p "terminado"
```

La leccion: un solo caracter faltante (una comilla) puede generar un error que parece estar en otro lado; cuando veas "unterminated string", busca la comilla de apertura sin su pareja, siempre hacia arriba desde donde marca el error.

Ejercicio 8: programa completo integrador. Escribe bienvenida.rb que: imprima un titulo con puts, pida el nombre con print (cursor en la misma linea), lea con gets.chomp, pida el lenguaje favorito, y despida con un mensaje que combine ambos datos usando +. Incluye al menos dos comentarios utiles. Solucion explicada:

```
# Programa de bienvenida - integra puts, print, gets y concatenacion
# Lo escribi para practicar el ciclo completo: pedir, leer y mostrar.

puts "=== Bienvenida a Ruby ==="
print "Como te llamas? "
nombre = gets.chomp
print "Cual es tu lenguaje favorito? "
lenguaje = gets.chomp # guardamos su respuesta para usarla abajo

# La despedida combina los dos datos con concatenacion simple
puts "Gracias, " + nombre + ". Que disfrutes " + lenguaje + " tanto como Ruby."
puts "Nos vemos en la Unidad 2."
```

Funciona porque print deja el cursor listo para la respuesta, gets.chomp lee limpio sin el salto de linea, y + une los strings en el orden dado. Los comentarios explican la intencion del programa y por que se guarda la variable lenguaje. Si al ejecutarlo el nombre sale con un renglon extra raro, revisa que hayas usado chomp y no gets solo.

Ejercicio 9: desafio de estilo. Reescribe este codigo "no idiomatico" en estilo Ruby elegante, explicando cada cambio.Codigo original:

```
def suma(a, b)
  resultado = a + b
  return resultado
end
puts(suma(3, 4))
```

Solucion explicada:

```
def suma(a, b)
  a + b
end
puts suma(3, 4)
```

Cambios: se elimino la variable intermedia resultado porque no aportaba nada (la suma ya es la ultima expresion); se quito return por el return implicito; se quitaron los parentesis de puts por ser un comando simple. El metodo paso de 4 lineas a 3 y se lee como una frase. Ojo: esto no es solo estetica; en equipos Ruby el codigo idiomático se revisa mas rapido y tiene menos bugs porque hay menos piezas que entender.

Resumen de la unidad

- Ruby es un lenguaje interpretado, dinamico y 100% orientado a objetos, creado por Yukihiro "Matz" Matsumoto en 1995 con un objetivo explicito: la felicidad del programador.
- La filosofia Ruby se resume en: todo es un objeto, principio de la menor sorpresa, mas de una forma de hacerlo, bloques expresivos y duck typing (importa lo que el objeto sabe hacer, no su clase).
- Instalacion: en Windows usa RubyInstaller con Devkit y verifica el PATH; en Mac usa rbenv sobre Homebrew y jamas toques el Ruby del sistema ni uses sudo para gemas; en Linux instala dependencias de compilacion y usa rbenv. En los tres casos verifica con `ruby -v`, `gem -v` e `irb`.
- `irb` es tu laboratorio: prueba ahi cualquier duda antes de escribir un archivo. Sal con `exit`.
- Tu primer programa es una linea: `puts "Hola, mundo"`, ejecutado con `ruby archivo.rb`. Sin ceremonias.
- La sintaxis elegante significa: sin punto y coma, indentacion de dos espacios, parentesis opcionales en comandos simples, return implicito, y nombres que documentan (`snake_case`, `CamelCase`, `MAYUSCULAS`, `?` y `!`).
- Los comentarios usan `#` y explican el POR QUE, no el QUE. Los comentarios que repiten el codigo son ruido y los que mienten son peores que ninguno.
- `puts` imprime con salto de linea, `print` imprime sin salto (ideal para pedir datos), `p` inspecciona mostrando la representacion real del objeto (tu aliado para depurar).
- Todo es un objeto: `5.class` es `Integer`, `"hola".upcase` funciona, `nil.nil?` es `true`. Esta idea es la llave maestra de todo el curso.
- El ciclo de trabajo es edita-guarda-ejecuta-lee el error-corrige-repite; lee los mensajes de error completos y prueba en `irb` antes de cambiar codigo al azar.

- Tabla mental de la unidad: Matz = felicidad del programador | RubyInstaller = Windows | rbenv = Mac/Linux profesional | irb = laboratorio | puts = mostrar | print = pedir en la misma linea | p = inspeccionar la verdad | # = comentario | return implicito = ultima expresion | ? = pregunta true/false | todo es objeto = no hay primitivos.

Tu editor de codigo: configuralo una vez, usalo siempre

Vas a pasar cientos de horas frente a tu editor, así que vale la pena configurarlo bien desde el primer día. El estándar de la industria para Ruby es VS Code (gratis, de Microsoft), aunque RubyMine (de pago, de JetBrains) es el favorito de muchos profesionales por su autocompletado inteligente y Sublime Text sigue teniendo fans por su velocidad. Cualquiera sirve; lo importante es la configuración.

Configuración mínima obligatoria para archivos .rb: indentación de dos espacios (en VS Code: ajustes, busca "tab size", pon 2, y activa "insert spaces"), resaltado de sintaxis para Ruby (VS Code lo trae o lo instalas como extensión "Ruby"), y la terminal integrada a la mano (en VS Code se abre con Ctrl + `). El flujo ideal: editor a la izquierda con tu código, terminal a la derecha abajo corriendo ruby archivo.rb. Guardas con Ctrl + S, subes a la terminal con un clic, presionas flecha arriba + Enter para repetir el último comando, ves el resultado. Ese ciclo de 5 segundos es tu ritmo de trabajo; si cada prueba te toma un minuto entre ventanas, tu aprendizaje va al 10% de velocidad.

Extensiones que valen oro: "Ruby" (resaltado y snippets), "RuboCop" (te marca errores de estilo mientras escribes, como un corrector ortográfico para código) y "Error Lens" (muestra los errores en la misma línea, sin tener que ir a la terminal). No instales 30 extensiones el primer día; estas tres bastan para empezar.

Buena práctica: crea una carpeta por unidad del curso (~/.ruby/unidad1, ~/.ruby/unidad2...) y guarda ahí tus programas con nombres descriptivos. Activa el guardado automático de tu editor para no perder trabajo (en VS Code: File > Auto Save). Y aprende los atajos de tu editor para duplicar líneas, mover líneas y comentar bloques (Ctrl + / comenta la selección en VS Code): te ahorran horas a la semana.

Error común: escribir código en el Bloc de notas de Windows sin resaltado ni numeración de líneas, y luego no encontrar donde marca el error. El mensaje dice "línea 14" y tu estás contando líneas a mano. Un editor decente te muestra los números de línea siempre: actívalos.

Como leer los errores de Ruby sin entrar en panico

Los errores no son tu enemigo, son mensajes de Ruby tratando de ayudarte. Aprende su idioma y depurar se vuelve un tramite en lugar de un drama. Un mensaje de error tipico tiene tres partes: el tipo de error, la descripcion y la pila (el rastro de lineas que llevaron al error).

Ejemplo real. Guarda esto como `error_demo.rb`:

```
nombre = "Ana"  
puts NOMBRE
```

Al ejecutarlo veras algo como:

```
error_demo.rb:2:in '<main>': uninitialized constant NOMBRE (NameError)
```

Traduccion linea por linea: `error_demo.rb:2` = el problema esta en la linea 2 de ese archivo. `'<main>'` = paso en el cuerpo principal del programa (no dentro de un metodo). `uninitialized constant NOMBRE` = usaste `NOMBRE` como si fuera una constante definida, pero no existe. `(NameError)` = el tipo de error: nombre desconocido. Diagnostico: escribiste `NOMBRE` en mayusculas (parece constante) cuando tu variable se llama `nombre` en minusculas. Ruby distingue mayusculas de minusculas siempre.

Los cinco errores que veras el 95% del tiempo en esta etapa:

1. `SyntaxError`: tu codigo no es Ruby valido (una comilla sin cerrar, un end faltante, un parentesis cojo). El interprete ni siquiera pudo leer el archivo. Casi siempre el problema esta ANTES de la linea que marca, asi que revisa hacia arriba.
2. `NameError`: usaste un nombre que no existe (variable con typo, constante no definida). Lee el nombre exacto del mensaje y buscalo en tu codigo.
3. `NoMethodError`: llamaste un metodo que el objeto no tiene, como `42.upcase` ("undefined method 'upcase' for 42"). Traduccion: "trataste un numero como si fuera texto". Revisa con `.class` que el objeto sea lo que crees.
4. `TypeError`: operacion entre tipos incompatibles, como `"hola" + 5` ("no implicit conversion of Integer into String"). Falta una conversion explicita (`to_s`, `to_i`).
5. `ArgumentError`: le pasaste al metodo un numero incorrecto de argumentos o un valor invalido (como `Integer("hola")`).

Estrategia profesional ante un error, en orden: lee el mensaje COMPLETO (no solo la primera linea), identifica el tipo y la linea, reproduce el pedazo sospechoso en irb aisladamente, corrige UNA cosa a la vez y vuelve a ejecutar. Lo que jamas debes hacer: cambiar cinco cosas a la

vez "a ver si jala", porque si funciona no sabras que lo arreglo y si empeora no sabras que lo rompio.

Analogia: un mensaje de error es como la luz del tablero de tu coche. El principiante la ignora o entra en panico; el conductor experimentado lee que dice ("presion baja en llanta delantera derecha"), revisa esa llanta y sigue. Ruby te esta diciendo exactamente que llanta revisar.

La comunidad Ruby: nunca programas solo

Una de las ventajas injustamente ignoradas de Ruby es su comunidad: famosa por ser acogedora con principiantes. Cuando te atores (y te vas a atorar, todos nos atoramos), estos son tus recursos, en orden de utilidad.

Primero, la documentacion oficial: ruby-doc.org tiene la referencia de cada clase y metodo con ejemplos. Cuando quieras saber que hace "hola".gsub o que metodos tiene Integer, ahi esta la respuesta autoritativa. Aprende a leerla desde ya: cada pagina muestra la clase, sus metodos y ejemplos cortos. Al principio se ve arida; a los dos meses sera tu pagina mas visitada.

Segundo, Stack Overflow y foros: el 99% de los errores de principiante ya los tuvo alguien antes. Copia el mensaje de error (sin tus datos personales) en el buscador y casi siempre hay una respuesta. Cuando preguntes tu, incluye: que querias lograr, el codigo minimo que reproduce el problema, el mensaje de error completo y que intentaste. Las preguntas bien hechas reciben respuesta en minutos; las de "no me jala, ayuda" se ignoran.

Tercero, la comunidad en espanol: hay grupos de Ruby en Telegram, Discord y Meetups en Mexico, Argentina, Colombia y Espana. Programar acompañado acelera todo: ves como otros resuelven, te corrigen vicios y haces contactos que después son oportunidades de trabajo. Lucas EGA Academy también es tu comunidad: comparte tus programas, pregunta sin pena.

Cuarto, lee codigo ajeno. Los repositorios de gemas populares en GitHub son clases magistrales gratis: ves como los profesionales nombran, organizan y documentan. Empieza por gemas pequenas y legibles; no trates de leer Rails completo el primer mes (es como intentar leer el Quijote en tu primera semana de espanol).

Buena practica: cuando resuelvas un error que te costo mas de 30 minutos, escríbelo en un archivo `notas_errores.md`: que decia el error, que era en realidad, como lo arreglaste. En seis meses tendras tu propio manual de depuracion personalizado, y cuando un companero se atore con lo mismo, tu seras quien lo ayude. Enseñar es la forma mas rapida de dominar.

Las versiones de Ruby: por que el numero importa

Cuando ejecutas `ruby -v` ves algo como 3.3.4. Ese numero no es decoracion: te dice que dialecto exacto del lenguaje tienes, y en el mundo profesional la version importa. Ruby usa versionado semantico: MAJOR.MINOR.PATCH. Cambios en PATCH (3.3.3 a 3.3.4) son correcciones seguras; cambios en MINOR (3.2 a 3.3) agregan funciones pero mantienen compatibilidad; cambios en MAJOR (2.x a 3.x) pueden romper cosas.

La historia rapida que debes conocer: Ruby 1.8 y 1.9 fueron la era clasica; 1.9 (2007) trajo velocidad y sintaxis moderna como los hashes `{ nombre: "Ana" }`. Ruby 2.x (2013-2020) fue la era de la madurez y el rendimiento. Ruby 3.0 (diciembre 2020) fue la gran promesa de Matz: "Ruby 3 sera 3 veces mas rapido que Ruby 2" (lo llamo 3x3), además de tipado estatico opcional con RBS y concurrencia real con Ractors y Fibers. Las versiones 3.x actuales son rapidas, estables y lo que encontraras en cualquier trabajo moderno.

Regla practica: para este curso y para empezar, cualquier Ruby 3.x sirve (3.0 en adelante). Si un tutorial usa sintaxis que tu Ruby no reconoce, el 90% de las veces es porque tu version es anterior a la del tutorial, no porque el tutorial este mal. En trabajo profesional, cada proyecto declara su version (en el Gemfile o en un archivo `.ruby-version`) y `rbenv/rvm` la respetan automaticamente al entrar a la carpeta. Por eso insistimos en el manejador de versiones: te permite tener 3.1 para el proyecto viejo del cliente y 3.3 para tu proyecto nuevo, sin conflictos.

Como saber si tu codigo usa algo de una version especifica: la documentacion de [ruby-doc.org](https://www.ruby-doc.org) indica desde que version existe cada metodo (veras notas como "New in version 2.7"). Acostumbrate a revisarlas cuando copies codigo de internet: un metodo "nuevo y elegante" de un blog de 2024 puede no existir en el Ruby 2.6 de un servidor viejo, y el error sera un confuso `NoMethodError`.

Buena practica: al inicio de tus notas o de tu README personal, anota con que version trabajas ("Practicado con Ruby 3.3.4"). Cuando algo no funcione y pidas ayuda, la version es el primer dato que te van a pedir, junto con el sistema operativo y el mensaje de error completo. Dar esos tres datos de entrada te ahorra dos rondas de preguntas.

Tu primer script util: que Ruby trabaje para ti

Ya dominas lo suficiente para escribir un script que haga algo UTIL en tu vida real, no solo ejercicios. La gracia de Ruby como lenguaje de scripting es que con 20 lineas automatizas tareas que a mano toman media hora. Aqui va un ejemplo completo, explicado, que organiza archivos por extension: guarda como `organizador.rb` y pruébalo en una carpeta de prueba (crea una carpeta con archivos falsos primero, no lo corras directo en tus Descargas reales hasta entenderlo).

```

# Organizador de archivos por extension
# Uso: ruby organizador.rb
# Mueve cada archivo de la carpeta actual a una subcarpeta según su extension.

require 'fileutils' # libreria estandar para mover archivos

Dir.glob("*").each do |archivo|
  next if File.directory?(archivo) # ignora subcarpetas
  next if archivo == "organizador.rb" # no te muevas a ti mismo

  extension = File.extname(archivo) # ".jpg", ".pdf", etc.
  extension = "sin_extension" if extension.empty?

  carpeta = extension.delete(".") # "jpg", "pdf"...
  FileUtils.mkdir_p(carpeta) # crea la carpeta si no existe
  FileUtils.mv(archivo, carpeta) # mueve el archivo
  puts "Movido: #{archivo} -> #{carpeta}/"
end

puts "Listo. Archivos organizados."

```

Que esta pasando, pieza por pieza: require 'fileutils' carga la libreria estandar para operaciones de archivos (viene con Ruby, no hay que instalar nada). Dir.glob("*") lista todo lo de la carpeta actual. each recorre la lista (lo veremos a fondo en la unidad 10; por ahora leelo como "para cada archivo, haz lo siguiente"). next salta a la siguiente iteracion cuando el archivo es una carpeta o el propio script (truco esencial: sin esa linea, el script se moveria a si mismo). File.extname extrae la extension; delete(".") quita el punto para usarla como nombre de carpeta. mkdir_p crea la carpeta (la p significa "sin quejarse si ya existe") y mv mueve. El puts final te muestra el progreso: en scripts, informar lo que haces es buena educacion.

Este script usa ideas de unidades futuras (bloques, File, Dir) como receta, pero el 80% es unidad 1 y 2: variables, strings, interpolacion, puts. Y demuestra la filosofia Ruby en accion: leer 20 lineas y entender que hace el programa sin manual. Modificalo como reto: haz que en lugar de mover, solo INFORME que moveria (cambia FileUtils.mv por un puts "Moveria..."), que es como los profesionales prueban scripts destructivos: primero en modo simulacro.

Moraleja de la unidad 1: ya no eres alguien que "quiere aprender a programar"; eres alguien con Ruby instalado, un editor configurado, tres comandos de salida dominados y un script util en su haber. Eso, colega, ya es programar.

Como estudiar este curso: practica deliberada (no solo leer)

Leer este libro sin programar es como aprender a nadar leyendo sobre natacion: entiendes la teoria y te ahogas en la practica. La investigacion sobre aprendizaje es clara: la practica deliberada (ejercicios enfocados, con retroalimentacion inmediata) avanza 10 veces mas rapido que la lectura pasiva. Aqui esta el metodo que recomiendo para cada unidad.

Paso 1: lee la explicacion con irb abierto. Cada vez que veas un ejemplo de codigo, ESCRIBELO tu mismo en irb o en un archivo y ejecutalo. No copies y pegues: el acto fisico de escribir fija la sintaxis en tu memoria muscular. Modifica los valores y predice el resultado antes de presionar Enter; cuando tu prediccion falle, ahi esta el aprendizaje real.

Paso 2: haz TODOS los ejercicios sin mirar la solucion primero. Atorarte 20 minutos en un ejercicio vale mas que leer 10 soluciones: tu cerebro construye los caminos neuronales en el esfuerzo, no en la respuesta. Si después de 30 minutos honestos no sale, lee la solucion, cierrala y reescribela de memoria. Si no puedes reescribirla sin mirar, no la entendiste todavia: repite.

Paso 3: el proyecto de cierre. Al terminar cada unidad, escribe un programita propio que combine lo aprendido con algo de TU vida: una calculadora de tus gastos, un organizador de tus series, un generador de excusas para tu jefe (es broma... o no). Los programas personales se quedan en la memoria porque te importan; los ejercicios genericos se olvidan.

Paso 4: repasa con espaciado. El olvido es exponencial: si no repasas, en una semana pierdes el 70%. La tecnica es simple: al empezar cada sesion de estudio, dedica 10 minutos a reescribir de memoria un ejercicio de la unidad anterior. Ese pequeno esfuerzo multiplica la retencion.

Errores de estudio comunes: el "tutorial hell" (ver curso tras curso sin escribir codigo propio), compararte con programadores de 10 anos de experiencia, y querer memorizar en lugar de entender (nadie memoriza todos los metodos de String; los profesionales buscan en la documentacion 20 veces al dia). Tu meta no es saberlo todo de memoria: es saber que existe, donde buscarlo y como combinarlo.

Glosario de la unidad 1

Interprete: programa (ruby) que lee tu codigo y lo ejecuta linea por linea, sin compilacion previa.

Sintaxis: las reglas de escritura del lenguaje; como la gramatica del espanol pero para codigo.

Puts / print / p: comandos para mostrar informacion; puts con salto de linea, print sin salto, p mostrando la representacion tecnica (inspect).

Comentario: texto ignorado por el interprete, marcado con #, que explica el por que del codigo a los humanos.

irb: consola interactiva de Ruby (Interactive Ruby); laboratorio para probar codigo al instante.

Gema (gem): libreria reutilizable de Ruby que instalas con gem install; el ecosistema tiene cientos de miles.

Variable: nombre que guarda un valor para usarlo después (se estudia a fondo en la unidad 2).

Metodo: accion que un objeto sabe hacer; se invoca con punto, como "hola".upcase.

Objeto: la unidad fundamental de Ruby; todo dato es un objeto con clase y metodos.

Clase: el molde o categoria de un objeto; 42 es de la clase Integer.

Snake_case: convencion de nombres en minusculas_con_guiones_bajos para variables y metodos.

CamelCase: convencion de nombres ComoEste para clases y modulos.

Return implicito: Ruby devuelve automaticamente la ultima expresion de un metodo, sin escribir return.

Principio de la menor sorpresa: filosofia de diseno de Ruby; el lenguaje debe comportarse como esperas.

Duck typing: "si camina como pato..."; importa lo que el objeto sabe hacer, no su clase.

RuboCop: herramienta que revisa tu estilo de codigo automaticamente según las convenciones de la comunidad.

PATH: lista de carpetas donde el sistema busca programas; Ruby debe estar ahi para que ruby funcione en cualquier terminal.

rbenv: manejador de versiones de Ruby para Mac/Linux; permite tener varias versiones sin conflictos.

RubyInstaller: instalador oficial de Ruby para Windows, con Devkit para compilar gemas.

Checklist final: verifica que dominas la unidad 1

Antes de pasar a la unidad 2, verifica honestamente cada punto. Si alguno te cuesta, regresa a su seccion: 10 minutos de repaso ahora valen horas después.

1. Puedo explicar con mis palabras quien es Matz y cual es la filosofia central de Ruby (la felicidad del programador). Si solo recuerdas "un japonés la creo", repasa la historia: la filosofia explica el 80% de las decisiones del lenguaje.
2. Tengo Ruby instalado y lo compruebo: `ruby -v`, `gem -v` e `irb` funcionan en mi terminal. Se que version tengo y en que sistema estoy.
3. Escribo, guardo y ejecuto un programa `.rb` sin mirar las instrucciones: creo el archivo, escribo `puts "hola"`, guardo y corro `ruby archivo.rb`.
4. Se la diferencia entre `puts`, `print` y `p`, y elijo el correcto según la situacion: `puts` para mostrar resultados, `print` para pedir datos en la misma linea, `p` para inspeccionar valores reales al depurar.
5. Escribo comentarios utiles con `#` que explican el por que, no el que. Puedo mirar un comentario mio de hace una semana y entender la decision.
6. Reconozco el return implicito y las convenciones de nombres: `snake_case` para variables y metodos, `CamelCase` para clases, `MAYUSCULAS` para constantes, `?` para preguntas y `!` para metodos que modifican.
7. Ante un error, leo el mensaje completo, identifico el tipo (`SyntaxError`, `NameError`, `NoMethodError`, `TypeError`, `ArgumentError`) y pruebo el pedazo sospechoso en `irb` antes de cambiar codigo al azar.
8. Tengo mi editor configurado: indentacion de dos espacios, numeros de linea visibles, terminal a la mano, y mis archivos organizados en carpetas por unidad con nombres descriptivos.
9. Se donde pedir ayuda: documentacion en `ruby-doc.org`, buscadores con el mensaje de error exacto, y como preguntar bien (que queria, codigo minimo, error completo, que intente).
10. Hice todos los ejercicios de la practica sin copiar las soluciones, y escribi al menos un programa propio que me sirva en mi vida real.

Si marcaste los 10, felicidades: tienes cimientos solidos. La unidad 2 construye directo sobre ellos: cada variable que declares y cada dato que manipules usara `puts`, `p`, `irb` y el ciclo edita-ejecuta que acabas de dominar.

UNIDAD 2: Variables y datos

En la unidad 1 instalaste Ruby y escribiste tus primeros programas; ahora vamos a construir el vocabulario fundamental con el que se escribe TODO lo demás: las variables y los datos. Piensa en esta unidad como aprender las palabras y la gramática básica de un idioma antes de intentar escribir un ensayo. Vas a dominar las variables y sus convenciones de nombres (porque en Ruby el nombre que eliges es documentación), los números enteros y decimales con sus operaciones (incluyendo las trampas famosas de la división entera que han hecho llorar a generaciones de principiantes), los strings con sus dos tipos de comillas y sus diferencias reales (no es un capricho estético, cambia lo que el intérprete hace), los símbolos y por qué existen (uno de los conceptos más "rubiescos" y más malentendidos), cómo preguntarle a cualquier dato de qué tipo es con `.class`, y cómo recibir datos del usuario con `gets.chomp` y convertirlos con `to_i`, `to_f` y `to_s`. Esta unidad corresponde a las clases 4, 5 y 6 del curso y es, sin exagerar, la que más vas a usar en tu vida como programador: el 90% del código que escribas será guardar datos en variables, transformarlos y mostrarlos. Si dominas estos seis temas, las unidades siguientes (condicionales, bucles, arreglos) se van a sentir naturales, porque todas manipulan variables y datos. Si los dejas flojos, todo lo demás se va a sentir como magia negra. Así que vamos con calma, con muchos ejemplos y con los errores típicos bien señalados para que no te atrapen.

Explicación detallada

Variables: cajitas con nombre y las reglas para nombrarlas

Una variable es un nombre que le pones a un valor para poder usarlo después. La analogía clásica es la cajita con etiqueta: creas una caja, le pegas una etiqueta que dice "edad" y dentro guardas el número 25. Cada vez que dices "edad" en tu código, Ruby abre la caja y te da lo que hay dentro. En Ruby crear una variable es trivial: escribes el nombre, un signo `=`, y el valor. No hay que declarar el tipo, no hay palabras reservadas previas, nada:

```
nombre = "Ana"
edad = 25
```

Listo, ya tienes dos variables. Pero aqui viene lo interesante: en Ruby las variables no son solo etiquetas, son parte del sistema de documentacion del codigo. El lenguaje define varios tipos de variables según el prefijo del nombre, y cada uno tiene un alcance (scope) distinto, es decir, desde donde se puede ver la variable. Los vas a conocer todos ahora porque confundirlos es fuente clasica de bugs.

Variables locales: empiezan con minuscula o guion bajo (nombre, edad, _temporal). Son las que mas vas a usar. Viven solo dentro del bloque o metodo donde se crean; fuera de ahi no existen. Son como notas en un post-it que dejas sobre tu escritorio: solo tu, en ese momento, las ves.

Variables de instancia: empiezan con @ (@nombre, @saldo). Pertenecen a un objeto y viven mientras el objeto viva. Las veremos a fondo en la unidad de clases y objetos; por ahora solo reconoce el @.

Variables de clase: empiezan con @@ (@@contador). Pertenecen a la clase completa y se comparten entre todos sus objetos. Se usan poco y con cuidado.

Variables globales: empiezan con \$ (\$config, \$debug). Se ven desde TODO el programa. Suenan tentadoras y son una trampa mortal: cualquier parte del codigo puede cambiarlas y cuando algo falla no sabes quien fue. La comunidad Ruby las considera mala practica casi siempre.

Constantes: empiezan con mayuscula (PI, IVA, MAX_INTENTOS). Su valor "no deberia" cambiar (Ruby te deja cambiarlo pero te regana con una advertencia). Se usan para valores fijos del negocio: tasas de impuestos, limites, URLs base.

Codigo malo (nombres que no dicen nada):

```
x = "Juan Perez"
y = 30
z = x + " tiene " + y.to_s + " anios"
puts z
```

Funciona, pero dentro de dos semanas ni tu vas a saber que eran x, y, z. Codigo bueno:

```
nombre_cliente = "Juan Perez"
edad_cliente = 30
mensaje = nombre_cliente + " tiene " + edad_cliente.to_s + " anios"
puts mensaje
```


El código bueno se lee solo. Regla de oro de la industria: un nombre de variable debe responder "que contiene" sin necesidad de leer el resto del código. `nombre_cliente` es mejor que `nc`, que es mejor que `x`. Los programadores profesionales prefieren nombres largos y claros sobre nombres cortos y cripticos; el autocompletado del editor escribe por ti, no hay excusa para ahorrar letras.

Convenciones de nombres en Ruby (memorízalas, son ley no escrita pero se cumplen como ley escrita): variables y métodos en `snake_case` (todas minúsculas, palabras separadas con guion bajo): `total_compra`, `calcular_iva`. Clases y módulos en `CamelCase` (cada palabra con mayúscula inicial, sin separadores): `CarritoCompra`, `Usuario`. Constantes en `MAYUSCULAS_SNAKE`: `TASA_IVA`, `INTENTOS_MAXIMOS`. Nada de `camelCase` para variables (`totalCompra`) ni de guiones medios (`total-compra`, que Ruby interpretaría como una resta). Si vienes de JavaScript donde el `camelCase` es rey, este es uno de esos ajustes culturales: en Ruby, verte escribir `totalCompra` es como llegar con tenis a una boda; nadie te corre, pero todos notan.

Otro detalle importante: en Ruby puedes reasignar una variable a un valor de otro tipo sin problema, porque las variables no tienen tipo fijo, solo los valores lo tienen:

```
dato = 42      # ahora dato guarda un Integer
dato = "hola"  # ahora guarda un String, sin quejarse
```

Esto se llama tipado dinámico y es una bendición y una responsabilidad: te da velocidad para prototipar, pero significa que TU debes llevar la cuenta de que hay en cada variable. Por eso los nombres claros importan el doble en lenguajes dinámicos.

Error común de principiante número 1: confundir `=` (asignación) con `==` (comparación). `dato = 5` guarda 5 en `dato`; `dato == 5` pregunta si `dato` vale 5. Escribir `if edad = 18` en lugar de `if edad == 18` asigna 18 a `edad` y el `if` siempre será verdadero (porque asignar devuelve el valor asignado, y 18 es "verdadero" en Ruby). Ruby te avisa con un warning en algunos casos, pero no siempre. Truco de veterano: lee `=` como "guarda" y `==` como "es igual a".

Error común número 2: usar una variable antes de asignarla. Ruby te lanza `NameError: undefined local variable`. Casi siempre significa un typo: escribiste `nmbre` en lugar de `nombre` en alguna línea. Lee el mensaje, busca el nombre exacto.

Error común número 3: pensar que las variables locales sobreviven entre archivos o entre ejecuciones. Cada vez que corres `ruby programa.rb`, todo empieza de cero; las variables no se guardan solas. Para persistencia necesitas archivos o bases de datos (unidades 23 en adelante).

Buena practica de la industria: inicializa tus variables con valores sensatos y cerca de donde las vas a usar. Evita las variables globales como la peste; si necesitas compartir estado, usa parametros, valores de retorno u objetos (lo veremos en su momento). Y adopta el habito de nombrar booleanos como preguntas: `activo`, `es_mayor`, `tiene_descuento`, en lugar de `bandera1` o `flag`. Cuando veas `if tiene_descuento`, tu cerebro lo lee sin esfuerzo.

Numeros: Integer, Float y las operaciones (con sus trampas famosas)

Ruby tiene dos tipos numericos basicos que debes dominar: Integer (enteros: ..., -2, -1, 0, 1, 2, ...) y Float (decimales: 3.14, -0.5, 2.0). A diferencia de otros lenguajes, en Ruby los enteros no se desbordan: si calculas un numero gigantesco, Ruby cambia automaticamente a una representacion de precision arbitraria. Prueba en irb **2 ¹⁰⁰⁰ (dos elevado a mil) y veras un monstruo de 302 digitos calculado sin quejarse. En C o Java eso seria un desbordamiento silencioso que te da un resultado erroneo; en Ruby simplemente funciona. Es un detalle hermoso del diseno.**

Las operaciones basicas son las que esperas: + suma, - resta, * multiplicacion, / division, % modulo (el residuo), **potencia. Pero hay TRES trampas famosas que debes conocer desde hoy porque muerden a todo principiante.**

Trampa 1: la division entera. En Ruby, si divides dos enteros, el resultado es entero (se trunca el decimal, no se redondea):

```
p 7 / 2    # 3, no 3.5
p 10 / 4   # 2, no 2.5
```

Esto sorprende a todos la primera vez. La logica historica: entero dividido entre entero da entero. Para obtener decimales, al menos uno de los dos debe ser Float:

```
p 7 / 2.0   # 3.5
p 7.0 / 2   # 3.5
p 7.to_f / 2 # 3.5 (to_f convierte a Float)
```

Codigo malo (el bug del principiante en calculadoras):

```
# Calcula el promedio de tres calificaciones... mal
promedio = (8 + 9 + 10) / 3
p promedio # 9, pero el promedio real es 9.0 y con otros datos se pierde precision
```

Con calificaciones 8, 7, 7: $(8+7+7)/3 = 22/3 = 7$ (entero), cuando el promedio real es 7.33. El alumno "pasa" o "reprueba" según un redondeo invisible. Código bueno:

```
promedio = (8 + 7 + 7).to_f / 3
p promedio # 7.333333333333333
```

Una sola conversión a Float al inicio contagia toda la operación. Acostumbrate: en cualquier división donde importen los decimales, convierte primero.

Trampa 2: los decimales Float no son exactos. Esto no es culpa de Ruby, es como las computadoras representan decimales en binario (punto flotante IEEE 754, el estándar mundial). Prueba en irb:

```
p 0.1 + 0.2 # 0.30000000000000004, no 0.3
```

Para la mayoría de los programas no importa, pero para DINERO sí importa y mucho: nunca representes dinero con Float en un sistema real, porque esos centavos fantasma se acumulan. La industria usa enteros para centavos (guarda 1999 en lugar de 19.99) o la clase BigDecimal para precisión exacta. Ruby incluye BigDecimal en su librería estándar; lo menciono para que lo conozcas, lo usaremos en ejemplos avanzados.

Trampa 3: el módulo % con negativos y la precedencia de operadores. El % devuelve el residuo: $10 \% 3 = 1$. Es útil para saber si un número es par ($n \% 2 == 0$) o para ciclos (turnos, paginación). Con negativos, Ruby sigue el signo del divisor, distinto a C: $-7 \% 3 = 2$ en Ruby. Y la precedencia: * y / se evalúan antes que + y -, como en matemáticas de primaria; usa parentesis cuando tengas duda, porque la claridad siempre gana:

```
p 2 + 3 * 4      # 14, no 20
p (2 + 3) * 4     # 20, parentesis mandan
```

Métodos numéricos útiles que debes conocer desde ya (recuerda: los números son objetos): even? y odd? (par/impar), abs (valor absoluto), round (redondea), ceil (redondea hacia arriba), floor (redondea hacia abajo), to_s (convierte a string), to_f (convierte a float), chr (código ASCII a carácter), times (repite un bloque n veces, lo veremos en bucles pero es un método de Integer). Ejemplos:

```
p 7.even?    # false
p 8.even?    # true
p (-5).abs   # 5
p 3.7.round  # 4
p 3.2.ceil   # 4
p 3.8.floor  # 3
```

Analogía: piensa en Integer y Float como dos tipos de vasos medidores en tu cocina. El Integer es el vaso de las tazas completas: solo mide porciones enteras. El Float es la jarra graduada con mililitros: mide fracciones. Si mezclas ambos en una receta (una operación), el resultado sale en la unidad más precisa (Float). Y la trampa de la división entera es como pedirle al vaso de tazas que mida media taza: te va a decir "una taza... o cero", porque no sabe de mitades.

Buena práctica de la industria: en código financiero, representa dinero en centavos como Integer (precio_centavos = 1999) y formatea solo al mostrar. En cálculos científicos donde la precisión decimal exacta importa, usa BigDecimal. En el resto, Float está bien. Y comenta SIEMPRE la unidad de tus variables numéricas: segundos o milisegundos, pesos o centavos, metros o centímetros. La mitad de los bugs numéricos del mundo real son confusiones de unidades, no errores de aritmética.

Strings: comillas simples, comillas dobles y sus diferencias reales

Un string es texto entre comillas, y en Ruby tienes dos tipos de comillas con comportamientos DISTINTOS. Esto no es decorativo: cambia lo que el intérprete hace con tu texto. Entender la diferencia es obligatorio.

Comillas dobles " ": son las "inteligentes". Dentro de ellas funcionan la interpolación (meter valores de variables con #{}) y las secuencias de escape (como \n para salto de línea, \t para tabulador):

```
nombre = "Ana"
puts "Hola, #{nombre}" # Hola, Ana (interpola el valor)
puts "línea1\nlínea2"  # imprime en dos líneas (\n = salto de línea)
```

Comillas simples ' ': son las "literales". Lo que escribes es lo que hay, sin interpolación ni escapes (salvo \\ y \\\n):

```
nombre = "Ana"
puts 'Hola, #{nombre}' # Hola, #{nombre} (literal, no interpola)
puts 'línea1\nlínea2'  # línea1\nlínea2 (literal, no salta de línea)
```

Codigo malo (el error que todos cometen una vez):

```
nombre = "Luis"
# Quiere saludar personalizado pero uso comillas simples:
puts 'Bienvenido, #{nombre}'
# Imprime literalmente: Bienvenido, #{nombre} (no lo que queria)
```

Codigo bueno:

```
nombre = "Luis"
puts "Bienvenido, #{nombre}"
# Imprime: Bienvenido, Luis
```

Regla practica: usa comillas dobles por defecto (es lo mas comun en codigo Ruby real), y comillas simples cuando quieras dejar claro que el texto es 100% literal, sin sorpresas. Algunos equipos usan simples para strings "estaticos" y dobles para los que interpolan, como documentacion visual; otros usan dobles siempre. Ambas escuelas son validas mientras seas consistente.

La interpolacion `#{}` es una de las joyas de Ruby: dentro puedes poner CUALQUIER expresion Ruby, no solo variables:

```
precio = 100
cantidad = 3
puts "Total: #{precio * cantidad} pesos" # Total: 300 pesos
puts "En mayusculas: #{'hola'.upcase}"   # En mayusculas: HOLA
```

Esto elimina la concatenacion tediosa con `+` y hace el codigo mucho mas legible. Compra mental: cada vez que sientas la tentacion de escribir `"texto " + variable + " mas texto"`, usa interpolacion en su lugar.

Concatenacion: aun asi debes conocerla. Con `+` unes strings (pero OJO: no puedes concatenar un string con un numero directamente, Ruby se queja: `TypeError`. Debes convertir primero con `to_s`). Con `<<` agregas al final del MISMO string (lo modifica, mas eficiente en bucles). Con `*` repites: `"ja" * 3 = "jajaja"`.

```
puts "hola" + " " + "mundo" # hola mundo
edad = 30
# puts "Tengo " + edad # ERROR: no se puede sumar String + Integer
puts "Tengo " + edad.to_s + " años" # bien: convertir primero
texto = "hola"
texto << "!"
p texto # "hola!" (el original cambio)
```

Métodos de string esenciales (los vas a usar toda la vida): length (o size, son alias: cuantos caracteres), upcase/downcase (mayúsculas/minúsculas), capitalize (primera en mayúscula), reverse (al revés), strip (quita espacios de los extremos), gsub (reemplaza todas las ocurrencias), split (divide en arreglo), include? (contiene?), empty? (vacío?). Pruebalos todos en irb ahora mismo, en serio, es la forma más rápida de fijarlos:

```
p "hola mundo".length      # 10
p "hola".upcase             # "HOLA"
p "HOLA".downcase          # "hola"
p "ana".capitalize         # "Ana"
p "hola".reverse           # "aloh"
p " hola ".strip           # "hola"
p "hola hola".gsub("hola", "adios") # "adios adios"
p "a,b,c".split(",")       # ["a", "b", "c"]
p "hola".include?("ol")    # true
p "".empty?                # true
```

Fíjate en el patrón: los métodos que terminan en ? preguntan (devuelven true/false) y los que terminan en ! (como gsub! o upcase!) modifican el string original en lugar de devolver uno nuevo. Ejemplo: texto.upcase devuelve una copia en mayúsculas y deja texto intacto; texto.upcase! convierte el propio texto. Esta convención ?/! es una de las cosas más elegantes de Ruby: con solo mirar el nombre sabes si el método pregunta o si es "peligroso" (modifica).

Analogía: los strings con comillas dobles son como cartas con espacios en blanco para rellenar ("Estimado _____, le informamos que su saldo es _____"), donde #{} rellena automáticamente. Los de comillas simples son como un texto ya impreso en piedra: lo que ves es lo que hay, nadie lo toca.

Error común: olvidar que los strings son MUTABLES en Ruby (se pueden cambiar), a diferencia de Python o Java donde son inmutables. Esto significa que pasar un string a un método que lo modifica con ! puede cambiar tu variable original sin que lo esperes. La defensa: usa las versiones sin ! cuando quieras conservar el original, y congela strings críticos si es necesario (tema avanzado). Otro error: comparar strings con == funciona por contenido

("hola" == "hola" es true), pero comparar con equal? compara identidad de objeto; para comparar texto usa siempre ==.

Buena practica: para textos largos o multilinea usa heredoc (lo veremos en la unidad de strings avanzados) o al menos divide con \ al final de linea. Y nunca construyas consultas SQL o comandos del sistema concatenando input del usuario sin sanitizar: es la puerta a inyecciones. Ese tema es de seguridad avanzada, pero el habito de desconfiar del input empieza aqui.

Simbolos: los primos ligeros de los strings (y por que existen)

Los simbolos son uno de los conceptos mas "rubiescos" que existen y también uno de los que mas confunden al principio, así que vamos despacio. Un simbolo se escribe con dos puntos al inicio: :nombre, :edad, :activo. Se parece a un string, se usa como un string en muchos lugares, pero NO es un string. Es otro tipo de dato, de la clase Symbol.

La diferencia fundamental: un simbolo es INMUTABLE y UNICO. Cada vez que escribes "hola" en tu codigo, Ruby crea un objeto String nuevo en memoria (aunque diga lo mismo). Cada vez que escribes :hola, Ruby usa EL MISMO objeto Symbol de siempre. Demostracion en irb:

```
p "hola".object_id # un numero, digamos 60
p "hola".object_id # OTRO numero distinto, digamos 80 (objeto nuevo)
p :hola.object_id  # un numero, digamos 123
p :hola.object_id  # EL MISMO 123 (mismo objeto reutilizado)
```

El object_id es como el numero de serie del objeto: si dos cosas tienen el mismo id, son el mismo objeto. Los strings "hola" crean gemelos identicos pero distintos; los simbolos :hola son siempre la misma persona.

Por que importa esto: rendimiento y memoria. Si tu programa usa la palabra "nombre" como etiqueta diez mil veces (por ejemplo, como clave en hashes o como nombre de metodo), con strings crearias diez mil objetos; con simbolos, uno solo. En aplicaciones grandes (como Rails, que usa simbolos por todos lados para nombres de rutas, parametros y opciones), la diferencia es enorme. Además, comparar simbolos es rapidisimo (se compara el id interno) mientras comparar strings recorre caracter por caracter.

Analogia: los strings son como tarjetas de presentacion impresas: cada vez que necesitas una, imprimes una nueva, aunque diga lo mismo. Los simbolos son como tu credencial de la empresa: hay una sola, la muestras mil veces y todos saben que eres tu. Para etiquetas, identificadores y nombres internos (cosas que se repiten y no cambian), la credencial (simbolo) es lo correcto. Para texto que el usuario lee, que cambia, que concatenas (mensajes, nombres, descripciones), la tarjeta impresa (string) es lo correcto.

Cuando usar cada uno, regla practica: usa simbolos para NOMBRES INTERNOS (claves de hashes, nombres de metodos, opciones, estados como :activo o :pendiente) y strings para TEXTO (mensajes al usuario, contenido que se muestra, datos que vienen de fuera). Ejemplo:

```
# Configuracion interna: simbolos
usuario = { nombre: "Ana", edad: 25, activo: true }
#           ^ claves como simbolos (etiquetas internas)

# Texto para el usuario: strings
puts "Bienvenida, #{usuario[:nombre]}"
```

Nota la sintaxis nombre: "Ana" dentro del hash: es el atajo moderno para :nombre => "Ana". Los hashes con claves de simbolo son TAN comunes en Ruby que el lenguaje creo una sintaxis especial para ellos. Lo veremos a fondo en la unidad 5; por ahora quedate con la idea.

Codigo malo (strings donde deberian ir simbolos):

```
estado = "activo"
if estado == "activo"
  puts "cuenta habilitada"
elsif estado == "suspendido"
  puts "cuenta pausada"
end
```

Funciona, pero cada "activo" es un objeto nuevo y si en algun lado escribes "Activo" con mayuscula por error, la comparacion falla silenciosamente. Codigo bueno:

```
estado = :activo
if estado == :activo
  puts "cuenta habilitada"
elsif estado == :suspendido
  puts "cuenta pausada"
end
```

Con simbolos hay un solo :activo posible; no hay riesgo de variantes con mayusculas escondidas y la comparacion es mas rapida. Además, cuando veas :activo en codigo ajeno, sabes al instante que es una etiqueta interna, no texto para mostrar.

Los simbolos también se convierten facil: `:hola.to_s` da `"hola"` y `"hola".to_sym` da `:hola`. Y responden a metodos utiles como `upcase` (`:hola.upcase` da `:HOLA`) y `length`. Pero OJO: no puedes interpolar dentro de un simbolo literal (`:hola_#{nombre}` no funciona como esperas en versiones viejas; en Ruby moderno si hay interpolacion en simbolos con `:"hola_#{nombre}"`, pero es un truco avanzado que rara vez necesitas).

Error comun de principiante: tratar de modificar un simbolo como si fuera string (`:nombre << "x"` falla, los simbolos son inmutables) o usar strings como claves de hash y luego buscar con simbolos (`hash["nombre"]` no es lo mismo que `hash[:nombre]`; son claves distintas y la busqueda devuelve nil, un bug silencioso y frustrante). La defensa: se consistente; en hashes modernos usa siempre la sintaxis `nombre: valor` (claves de simbolo) y accede con `[:nombre]`.

Buena practica de la industria: en Rails y en la mayoria de las gemas, las opciones de configuracion son simbolos (por ejemplo, `render json: datos, status: :ok`). Acostumbrate a leer `:ok`, `:created`, `:not_found` como vocabulario estandar: son los estados HTTP expresados como simbolos. Cuando disenes tus propios metodos con opciones, usa simbolos para los valores fijos y documentalos.

Tipos de datos y .class: preguntale a Ruby que es cada cosa

Ya conoces Integer, Float, String y Symbol. La familia completa de tipos basicos incluye además: TrueClass y FalseClass (los valores true y false), NilClass (el valor nil, la "nada"), Array (listas ordenadas), Hash (pares clave-valor), Range (rangos como 1..10). Las unidades siguientes cubren Array, Hash y Range a fondo; TrueClass/FalseClass/nil los usaremos desde ya en condicionales.

El metodo `.class` te dice a que clase pertenece cualquier objeto. Es tu detector de tipos universal:

```
p 42.class      # Integer
p 3.14.class    # Float
p "hola".class  # String
p :hola.class   # Symbol
p true.class    # TrueClass
p false.class   # FalseClass
p nil.class     # NilClass
p [1, 2].class  # Array
p({ a: 1 }).class # Hash
p (1..5).class  # Range
```

Por que te debe importar: muchos errores de principiante son errores de tipo disfrazados. "undefined method 'upcase' for 42" significa "trataste de usar un metodo de String sobre un Integer". Cuando veas un error asi, tu primer reflejo debe ser preguntar `p variable.class` para confirmar que la variable contiene lo que crees. El 90% de las veces el problema es que un dato llego como String cuando esperabas Integer (spoiler: gets siempre devuelve String, lo vemos en la siguiente seccion).

También existen los metodos de pregunta `is_a?` y `kind_of?` (son alias, hacen lo mismo): te dicen si un objeto pertenece a una clase o a sus ancestros:

```
p 42.is_a?(Integer) # true
p 42.is_a?(Numeric) # true (Integer hereda de Numeric)
p "hola".is_a?(Integer) # false
```

Y `respond_to?` te dice si un objeto sabe hacer algo (duck typing en accion):

```
p "hola".respond_to?(:upcase) # true
p 42.respond_to?(:upcase)      # false
p 42.respond_to?(:even?)      # true
```

Estos metodos son las herramientas del programador defensivo: antes de operar con un dato que viene de fuera (del usuario, de un archivo, de internet), verificas que sea lo que esperas o que responda a lo que necesitas. En codigo profesional veras mucho `respond_to?` y conversiones explicitas; es lo que separa un script de juguete de software confiable.

Analogia: `.class` es como preguntarle a alguien "a que te dedicas?" y `respond_to?` es como preguntar "sabes manejar?". Para algunas tareas te importa el titulo (la clase), pero para la mayoria te importa la habilidad (los metodos). Ruby, con su filosofia duck typing, prefiere preguntar por habilidades.

Error comun: usar `==` para comparar clases (`42.class == Integer` funciona, pero es rigido) en lugar de `is_a?`, que respeta la herencia. Si defines una clase `Empleado` que hereda de `Persona`, `empleado.is_a?(Persona)` es true pero `empleado.class == Persona` es false. En codigo con herencia (unidad 7), `is_a?` es casi siempre lo correcto.

Tabla mental de tipos basicos: Integer (enteros, sin limite), Float (decimales, cuidado con la precision), String (texto, mutable, comillas), Symbol (etiqueta inmutable y unica, dos puntos), true/false (booleanos), nil (ausencia de valor, el "nada"), Array (lista ordenada [...]), Hash (diccionario {clave: valor}), Range (intervalo 1..10). Todo responde a `.class` y todo es un objeto.

Entrada de datos con gets: tu programa escucha al usuario

Hasta ahora tus programas hablan pero no escuchan: muestran informacion pero no reciben nada. gets cambia eso: es el metodo que lee una linea de lo que el usuario escribe en el teclado. El nombre viene de "get string" (obtener cadena), y el nombre ya te avisa de la trampa: SIEMPRE devuelve un String, aunque el usuario escriba un numero.

Programa minimo:

```
puts "Como te llamas?"
nombre = gets
p nombre
```

Si escribes Ana y presionas Enter, p muestra "Ana\n": el string incluye el salto de linea del Enter. Ese \n escondido es fuente de bugs sutiles (comparaciones que fallan, concatenaciones con renglones extra). La solucion es chomp, que "muerde" (chomp = masticar) el salto de linea final:

```
puts "Como te llamas?"
nombre = gets.chomp
p nombre # "Ana", limpio
```

gets.chomp es TAN comun que lo vas a escribir miles de veces; algunos programadores lo consideran una sola palabra. Memorizalo como receta: para leer texto limpio, siempre gets.chomp.

Que pasa tecnicamente: gets pausa tu programa y espera. El usuario escribe y presiona Enter. Ruby toma todo lo escrito (incluido el \n del Enter) y lo devuelve como String. chomp devuelve una copia sin el separador de linea final. Si el usuario solo presiona Enter sin escribir nada, gets.chomp devuelve "" (string vacio), no nil. Y si la entrada se cierra (por ejemplo, redirigiendo un archivo que se acaba), gets devuelve nil: en scripts robustos se verifica.

Codigo malo (olvidar chomp y sufrir):

```
print "Escribe si o no: "
respuesta = gets
if respuesta == "si"
  puts "Dijiste si"
else
  puts "No te entendi" # SIEMPRE cae aqui, porque respuesta es "si\n"
end
```

El usuario escribe "si", pero respuesta contiene "si\n" y la comparacion falla. Es un bug invisible a simple vista.Codigo bueno:

```
print "Escribe si o no: "
respuesta = gets.chomp
if respuesta == "si"
  puts "Dijiste si"
else
  puts "No te entendi"
end
```

Un solo .chomp y el programa funciona. Este bug ha hecho perder horas a mas programadores de los que admitirian; ahora tu ya estas vacunado.

Variantes utiles: gets.chomp es la base, pero también existe gets.strip (quita espacios de ambos extremos además del salto de linea, util cuando el usuario deja espacios accidentales: " Ana " -> "Ana") y el patron gets.chomp.downcase para normalizar respuestas ("SI", "Si", "si" todas se vuelven "si"). Para programas que piden varias cosas, el patron se repite:

```
print "Nombre: "
nombre = gets.chomp.strip
print "Ciudad: "
ciudad = gets.chomp.strip
puts "#{nombre} vive en #{ciudad}"
```

Buena practica: valida la entrada, no confies en ella. Si pides una edad y el usuario escribe "veinte", tu programa debe manejarlo con elegancia (lo veremos con conversion y validacion en la siguiente seccion y en la unidad de excepciones). Como minimo, verifica que el string no este vacio antes de usarlo:

```
print "Tu nombre (no puede estar vacio): "
nombre = gets.chomp.strip
if nombre.empty?
  puts "No escribiste nada. Intenta de nuevo."
else
  puts "Hola, #{nombre}"
end
```

Analogia: gets es como un microfono abierto en un programa de radio: cualquiera puede decir cualquier cosa, incluyendo disparates. Tu trabajo como programador es ser el conductor que escucha, limpia (chomp/strip) y verifica antes de ponerlo al aire. Un buen programa nunca repite al aire lo que el microfono capto sin revisarlo.

Conversion de tipos: to_i, to_f, to_s (y sus primos)

Aqui esta la pieza que une todo: como gets siempre devuelve String, necesitas convertir ese texto a numero para calcular, y a veces convertir numeros a texto para mostrarlos. Ruby te da metodos de conversion con nombres predecibles: to_i (a Integer), to_f (a Float), to_s (a String). Se leen "to integer", "to float", "to string".

```
edad_texto = "25"
edad = edad_texto.to_i
p edad      # 25 (Integer)
p edad.class # Integer

precio_texto = "19.99"
precio = precio_texto.to_f
p precio # 19.99 (Float)

numero = 42
texto = numero.to_s
p texto      # "42" (String)
p texto.class # String
```

El patron completo de pedir un numero al usuario, que usaras en cientos de programas:

```
print "Cuantos anios tienes? "
edad = gets.chomp.to_i
puts "El anio que viene tendras #{edad + 1}"
```

gets.chomp.to_i encadena tres pasos: leer, limpiar, convertir. Si el usuario escribe "25", edad vale 25 (Integer) y edad + 1 funciona. Sin el to_i, "25" + 1 daria error (no se puede sumar String + Integer) y "25" + "1" daria "251" (concatenacion, no suma). Esta es LA trampa numero uno de los programas interactivos de principiantes; si tu calculadora "suma" 2 + 2 y da 22, ya sabes que olvidaste convertir.

Comportamiento con texto no numerico (importante): to_i y to_f son tolerantes: convierten lo que pueden desde el inicio y si no hay nada numerico, devuelven 0:

```
p "25 años".to_i # 25 (toma el número inicial)
p "19.99".to_i   # 19 (trunca el decimal)
p "19.99".to_f   # 19.99
p "hola".to_i    # 0 (no hay número, devuelve 0 sin quejarse)
p "".to_i        # 0
p " 42 ".to_i    # 42 (ignora espacios)
```

Esa tolerancia es cómoda pero peligrosa: si el usuario escribe "veinte", `to_i` devuelve 0 sin avisar y tu programa calcula con un cero fantasma. En programas serios se valida antes de convertir (por ejemplo, con una expresión regular que verifique que el texto son solo dígitos, tema de la unidad 9) o se usa `Integer(texto)`, que es **ESTRICTO** y lanza un error si el texto no es un número válido:

```
p Integer("25")    # 25
p Integer("hola")  # ArgumentError: invalid value for Integer()
p Float("19.99")   # 19.99
p Float("hola")    # ArgumentError
```

Regla práctica: `to_i/to_f` para conversión rápida y tolerante en scripts y prácticas; `Integer()/Float()` con manejo de errores cuando el dato debe ser válido si o si (formularios, archivos, APIs). Lo segundo lo dominaremos en la unidad de excepciones.

`to_s` convierte casi cualquier cosa a texto: `42.to_s` es "42", `3.14.to_s` es "3.14", `true.to_s` es "true", `nil.to_s` es "" (string vacío, ojo con esto), `[1,2].to_s` es "[1, 2]". Y recuerda que la interpolación `#{} llama a to_s automáticamente`, por eso "Tienes #{edad} años" funciona aunque `edad` sea `Integer`: Ruby convierte por ti dentro de `#{}.`

Primos de la familia de conversión que debes conocer: `to_a` (a arreglo: `"hola".chars.to_a`, `(1..5).to_a` da `[1,2,3,4,5]`), `to_sym` (a símbolo: `"nombre".to_sym` da `:nombre`), `to_h` (a hash, para estructuras compatibles). No los memorices todos ahora; reconoce el patrón `to_` y sabrás buscar el que necesites en la documentación.

Código malo (conversión olvidada, el bug clásico):

```
print "Precio: "
precio = gets.chomp
print "Cantidad: "
cantidad = gets.chomp
total = precio * cantidad # ERROR o resultado absurdo
puts "Total: #{total}"
```

Si el usuario escribe 100 y 3, `precio * cantidad` intenta `"100" * "3"`: `String * String` no existe y explota. (Curiosidad: `"100" * 3` SI funciona y da `"100100100"`, porque `String * Integer` repite; el bug seria silencioso y peor.) Código bueno:

```
print "Precio: "  
precio = gets.chomp.to_f  
print "Cantidad: "  
cantidad = gets.chomp.to_i  
total = precio * cantidad  
puts "Total: $#{total}"
```

Con las conversiones explicitas, `100 * 3 = 300.0` correcto. Moraleja grabada a fuego: todo lo que viene de `gets` es `String` hasta que TU lo conviertas.

Buena practica de la industria: convierte en la frontera, es decir, justo donde el dato entra a tu programa (al leerlo con `gets`, al leerlo de un archivo, al recibirlo de la red). A partir de ahi, el resto de tu código trabaja con tipos correctos y no tiene que adivinar. Este principio se llama "parse, don't validate" en su version elegante, pero la version practica es: un dato sin convertir es una bomba de tiempo; desactivala en la entrada.

Juntando todo: el modelo mental de la unidad 2

Cierra los ojos y arma el mapa: tu programa vive en un mundo de OBJETOS (unidad 1). Cada objeto tiene una CLASE que puedes preguntar con `.class`. Guardas objetos en VARIABLES con nombres claros en `snake_case`. Los numeros son `Integer` o `Float` (cuidado con la division entera y la precision de `Float`). El texto son `STRINGS` (comillas dobles para interpolar, simples para literal) o `SYMBOLS` (etiquetas internas unicas e inmutables). Tu programa ESCUCHA con `gets.chomp` (siempre `String`) y CONVIERTE con `to_i/to_f/to_s` en la frontera. Con estas piezas puedes escribir cualquier programa de entrada-proceso-salida, que es el 90% de lo que haras como principiante y una buena parte de lo que haras como profesional.

Un ultimo consejo de colega antes de los ejemplos: cuando algo no funcione, el ritual de depuracion de esta unidad es siempre el mismo. Primero, `p` variable para ver el valor real. Segundo, `p variable.class` para ver el tipo real. Tercero, compara lo que ves con lo que esperabas. Nueve de cada diez bugs de esta etapa son "crei que era `Integer` pero era `String`" o "crei que valia 25 pero vale `'25\n'`". Con `p` y `.class` los encuentras en segundos.

Ejemplos paso a paso

Ejemplo 1: variables bien nombradas y reasignacion. Guarda como variables.rb. Muestra declaracion, uso en operaciones y que el tipo puede cambiar.

```
# Variables con nombres claros en snake_case
nombre_estudiante = "Sofia"
calificacion = 9.5
aprobado = true

puts "Estudiante: #{nombre_estudiante}"
puts "Calificacion: #{calificacion}"
puts "Aprobado: #{aprobado}"

# Las variables no tienen tipo fijo: pueden guardar otra cosa después
dato = 42
p dato.class # Integer
dato = "ahora soy texto"
p dato.class # String
```

La ultima parte demuestra el tipado dinamico: la misma variable guardo un Integer y luego un String. Los nombres con guion bajo se leen como frases.

Ejemplo 2: la trampa de la division entera, demostrada. Guarda como division.rb. Este ejemplo vale oro: ejecutalo y mira los resultados.

```
puts "Division entera (dos Integer):"
p 7 / 2      # 3 (trunca, no redondea)
p 10 / 4     # 2

puts "Division con decimales (al menos un Float):"
p 7 / 2.0     # 3.5
p 7.0 / 2     # 3.5
p 7.to_f / 2  # 3.5

puts "Promedio de calificaciones:"
suma = 8 + 7 + 7
p suma / 3     # 7 (mal: perdio el .33)
p suma.to_f / 3 # 7.333333333333333 (bien)
```

Cada p muestra el valor real. El ejemplo del promedio es el caso de uso donde esta trampa muerde en la vida real: un sistema escolar que trunca promedios.

Ejemplo 3: comillas simples vs dobles, laboratorio visual. Guarda como comillas.rb.

```
nombre = "Diego"
animal = "perro"

puts "Con comillas dobles (interpolan):"
puts "Hola, #{nombre}, tu #{animal} es bonito"

puts "Con comillas simples (literales):"
puts 'Hola, #{nombre}, tu #{animal} es bonito'

puts "Secuencias de escape:"
puts "Primera linea\nSegunda linea"    # \n salta de linea
puts 'Primera linea\nSegunda linea'    # literal: se ve el \n
```

Ejecuta y compara: las dobles sustituyen variables y entienden \n; las simples muestran todo tal cual. Este es el experimento que debes hacer una vez para no olvidarlo nunca.

Ejemplo 4: metodos de string encadenados. Guarda como strings_metodos.rb. Muestra los metodos esenciales y el encadenamiento.

```
frase = "  Aprendo Ruby en Lucas EGA Academy  "
p frase.strip                # quita espacios de los extremos
p frase.strip.upcase         # encadena: limpia y mayusculas
p frase.strip.downcase.capitalize # limpia, minusculas, primera mayuscula

p "manzana,pera,uva".split(",") # ["manzana", "pera", "uva"]
p "hola mundo".gsub("mundo", "Ruby") # "hola Ruby"
p "programar".reverse        # "ramargorp"
p "Ruby".length               # 4
```

El encadenamiento (un metodo tras otro) es estilo Ruby puro: cada metodo devuelve un string y el siguiente lo transforma. Leelo de izquierda a derecha como una tuberia.

Ejemplo 5: simbolos vs strings, la prueba del object_id. Guarda como simbolos.rb.

```
puts "Strings: cada literal es un objeto nuevo"
p "ruby".object_id
p "ruby".object_id
puts "Observa: los numeros son DISTINTOS"

puts "Simbolos: siempre el mismo objeto"
p :ruby.object_id
p :ruby.object_id
puts "Observa: los numeros son IGUALES"

# Uso tipico: simbolos como etiquetas, strings como texto
config = { idioma: "espanol", nivel: "principiante" }
puts "Idioma: #{config[:idioma]}, Nivel: #{config[:nivel]}"
```

La primera parte es la demostracion que debes ver con tus ojos. La segunda muestra el uso real: hash con claves de simbolo (sintaxis moderna idioma: "espanol") y acceso con [:idioma].

Ejemplo 6: .class e is_a? como detectores. Guarda como tipos.rb.

```
valores = [42, 3.14, "hola", :hola, true, nil, [1, 2]]
valores.each do |v|
  p v
  puts "  clase: #{v.class}"
end

puts "Preguntas de tipo:"
p 42.is_a?(Integer)    # true
p 42.is_a?(Numeric)    # true (herencia)
p "hola".respond_to?(:upcase) # true
p 42.respond_to?(:upcase)    # false
```

El bucle each recorre el arreglo (adelanto de la unidad 10, usalo como receta por ahora) y muestra que .class funciona sobre cualquier cosa. Las preguntas finales muestran is_a? y respond_to?.

Ejemplo 7: programa interactivo completo con conversion. Guarda como calculadora_edad.rb. El patron gets.chomp.to_i en accion real.

```

print "En que anio naciste? "
anio = gets.chomp.to_i
print "En que anio estamos? "
actual = gets.chomp.to_i

edad = actual - anio
p edad # inspeccionamos el numero calculado

puts "Tienes (o cumplirás) #{edad} años."
puts "En 10 años tendrás #{edad + 10}."

# Demostración del bug si olvidaras convertir:
puts "Si no hubieramos convertido, 'anio' seria String:"
p anio.class # Integer, gracias a to_i

```

Pruebalo con tu anio real de nacimiento. La ultima linea confirma que la conversion funciono. Cambia `to_i` por `to_s` y observa el desastre: es el mejor maestro.

Ejemplo 8: `to_s`, `to_f` y la tolerancia de `to_i`. Guarda como `conversiones.rb`.

```

puts "to_s convierte todo a texto:"
p 42.to_s      # "42"
p 3.14.to_s    # "3.14"
p true.to_s    # "true"
p nil.to_s     # "" (vacío, cuidado)

puts "to_i es tolerante (a veces demasiado):"
p "25 años".to_i # 25
p "hola".to_i    # 0 (no avisa, devuelve 0)
p "".to_i        # 0

puts "Integer() es estricto:"
p Integer("25") # 25
begin
  p Integer("hola")
rescue ArgumentError => e
  puts "Error capturado: #{e.message}"
end

```

La ultima parte usa `begin/rescue` para atrapar el error de `Integer("hola")` sin que el programa muera (adelanto de la unidad 8, usalo como receta). Muestra la diferencia entre conversion tolerante y estricta.

Ejemplo 9: mini sistema de tienda que integra TODA la unidad. Guarda como tienda.rb. Variables claras, numeros, strings, interpolacion, gets, conversion.

```
# Mini sistema de tienda - integra toda la Unidad 2
puts "=== Tienda Lucas EGA ==="
print "Nombre del producto: "
producto = gets.chomp.strip
print "Precio unitario: "
precio = gets.chomp.to_f
print "Cantidad: "
cantidad = gets.chomp.to_i

subtotal = precio * cantidad
iva = subtotal * 0.16 # TASA_IVA seria mejor como constante
total = subtotal + iva

puts "=== Ticket ==="
puts "Producto: #{producto.capitalize}"
puts "Precio: ${precio} x #{cantidad}"
p subtotal # verificamos el calculo exacto
puts "IVA (16%): ${iva.round(2)}"
puts "TOTAL: ${total.round(2)}"
puts "Gracias por su compra."
```

Este programa pide texto y numeros, convierte en la frontera, calcula con Float, redondea para mostrar con round(2) y usa p para verificar el subtotal exacto. Es un programa "de verdad" en 20 lineas. Mejora propuesta: extrae el 0.16 a una constante TASA_IVA al inicio.

Practica

Ejercicio 1: nombres que documentan. Declara variables para guardar: el nombre de tu ciudad, tu edad, si tienes mascota (true/false) y tu estatura en metros, usando snake_case y nombres descriptivos. Imprimelos con interpolacion. Solucion explicada:

```
nombre_ciudad = "Guadalajara"
edad = 28
tiene_mascota = true
estatura_metros = 1.75

puts "Vivo en #{nombre_ciudad}, tengo #{edad} anios."
puts "Mascota: #{tiene_mascota}, estatura: #{estatura_metros} m"
```

Funciona porque cada nombre dice exactamente que contiene (nombre_ciudad mejor que c, tiene_mascota se lee como pregunta) y la interpolacion `#{}` convierte automaticamente los numeros y booleanos a texto. Nota estatura_metros: incluir la unidad en el nombre previene el bug clasico de confundir metros con centimetros.

Ejercicio 2: division entera vs decimal. Calcula cuanto le toca a cada persona si 1000 pesos se reparten entre 6 personas, primero con division entera y luego con decimales. Explica la diferencia. Solucion explicada:

```
total = 1000
personas = 6
p total / personas          # 166 (entero: se pierden 4 pesos)
p total.to_f / personas    # 166.66666666666666 (exacto)
```

La primera da 166: al dividir dos Integer, Ruby trunca. En un reparto real esos 4 pesos "desaparecen" del calculo ($166 \times 6 = 996$). La segunda convierte a Float primero y da el valor exacto. Moraleja: en repartos de dinero, convierte a Float (o trabaja en centavos como Integer, la practica profesional) antes de dividir.

Ejercicio 3: comillas simples vs dobles. Predice que imprime cada linea y luego compruebalo en irb.Codigo:

```
nombre = "Marta"
puts 'Hola #{nombre}'
puts "Hola #{nombre}"
puts 'Ruta: C:\nuevo'
puts "Ruta: C:\nuevo"
```

Solucion explicada: la primera imprime `Hola #{nombre}` literal (simples no interpolan); la segunda imprime `Hola Marta`; la tercera imprime `Ruta: C:\nuevo` literal; la cuarta interpreta `\n` como salto de linea e imprime `"Ruta: C:"` en una linea y `"uevo"` en la siguiente (el `\n` parte el texto). Este ultimo caso es la trampa real: en rutas de Windows con comillas dobles, `\n` se vuelve salto de linea. Por eso en Windows se usan comillas simples para rutas o se escapa la barra: `"C:\\nuevo"`.

Ejercicio 4: metodos de string en cadena. Toma tu nombre completo en minusculas con espacios extra al inicio y final, y en UNA linea de codigo encadenado devuelvelo limpio y capitalizado correctamente (primera letra de cada palabra en mayuscula). Pista: necesitaras `split`, `map` y `join` (`map` lo veremos a fondo en la unidad 6, usalo como receta). Solucion explicada:

```
nombre = "  juan perez lopez  "
p nombre.strip.split(" ").map { |p| p.capitalize }.join(" ")
# "Juan Perez Lopez"
```

Paso a paso: strip quita los espacios externos ("juan perez lopez"), split(" ") lo parte en ["juan", "perez", "lopez"], map con capitalize convierte cada palabra (["Juan", "Perez", "Lopez"]) y join(" ") las une con espacios. El encadenamiento se lee como tubería de izquierda a derecha. Es código Ruby 100% idiomático y es exactamente como se limpian nombres en sistemas reales.

Ejercicio 5: símbolos como estados. Escribe un programa que guarde el estado de un pedido en una variable (usa un símbolo: :recibido, :enviado o :entregado) e imprima un mensaje distinto según el estado con if/elsif. Solución explicada:

```
estado_pedido = :enviado

if estado_pedido == :recibido
  puts "Tu pedido fue recibido, lo estamos preparando."
elsif estado_pedido == :enviado
  puts "Tu pedido va en camino."
elsif estado_pedido == :entregado
  puts "Pedido entregado. Gracias por tu compra."
end
```

Usamos símbolos porque son etiquetas internas fijas: hay un solo :enviado posible, la comparación es rápida y nadie va a escribir "Enviado" con mayúscula por accidente. Cambia el valor inicial a :recibido o :entregado y observa como cambia el mensaje. (En la unidad 8 verás case/when, que es la forma elegante de escribir estas cadenas de comparación.)

Ejercicio 6: detective de tipos. Sin ejecutar, predice la clase de cada expresión y luego verifica con .class en irb: 5 + 2, 5 / 2, 5 / 2.0, "5".to_i, "hola".to_sym, nil.to_s, (1..3).to_a. Solución explicada: 5 + 2 es Integer (entero+entero); 5 / 2 es Integer (división entera trunca, da 2); 5 / 2.0 es Float (si un operando es Float, el resultado es Float); "5".to_i es Integer; "hola".to_sym es Symbol; nil.to_s es String (string vacío ""); (1..3).to_a es Array ([1, 2, 3]). La lección: el tipo del resultado depende de los tipos de entrada y de la operación; ante la duda, pregunta con .class en lugar de adivinar.

Ejercicio 7: calculadora de propina interactiva. Pide el total de la cuenta y el porcentaje de propina, calcula propina y total, y muestra todo redondeado a 2 decimales. Solución explicada:

```

print "Total de la cuenta: "
cuenta = gets.chomp.to_f
print "Porcentaje de propina (ej. 15): "
porcentaje = gets.chomp.to_f

propina = cuenta * (porcentaje / 100)
total = cuenta + propina

puts "Propina: ${propina.round(2)}"
puts "Total a pagar: ${total.round(2)}"

```

Convertimos a Float en la frontera con `to_f` porque el dinero tiene centavos. `porcentaje / 100` convierte "15" en 0.15. `round(2)` deja dos decimales para mostrar pesos y centavos. Si el usuario escribe "diez" en vez de 10, `to_f` devuelve 0.0 y la propina sale en 0: es la tolerancia silenciosa de `to_f`; en un sistema real validaríamos (unidades 22 y 27).

Ejercicio 8: el bug de la concatenacion. Este programa falla; explica por que y corrígelo de dos formas distintas (con `to_s` y con interpolacion). Código con error:

```

print "Tu edad: "
edad = gets.chomp
puts "El año que viene tendrás " + edad + 1

```

Solución explicada: hay DOS errores encadenados. Primero, `edad` es String (`gets.chomp`) y `"texto" + 1` lanza `TypeError` (no se puede sumar String e Integer). Segundo, aunque `edad` fuera número, la precedencia haría `("...tendrás " + edad) + 1`, que seguiría mal. Corrección con `to_s`:

```

edad = gets.chomp.to_i
puts "El año que viene tendrás " + (edad + 1).to_s

```

Corrección con interpolación (preferida):

```

edad = gets.chomp.to_i
puts "El año que viene tendrás #{edad + 1}"

```

La versión con interpolación es mejor: convierte `edad` a Integer en la frontera, calcula `edad + 1` como números, y `#{}` lo convierte a texto automáticamente. Una línea clara en lugar de concatenaciones con parentesis.

Ejercicio 9: desafio integrador, conversor de temperaturas. Pide una temperatura en Celsius, conviértela a Fahrenheit y Kelvin, y muestra las tres con 1 decimal. Incluye comentarios que expliquen las formulas. Solucion explicada:

```
# Conversor de temperaturas - integra gets, to_f, operaciones e interpolacion
# Formulas: F = C * 9/5 + 32 | K = C + 273.15

print "Temperatura en Celsius: "
celsius = gets.chomp.to_f

# OJO: 9/5 con enteros daria 1 (division entera); usamos 9.0/5 para forzar Float
fahrenheit = celsius * 9.0 / 5 + 32
kelvin = celsius + 273.15

puts "#{celsius} C son:"
puts "  #{fahrenheit.round(1)} F"
puts "  #{kelvin.round(1)} K"
```

Los puntos clave: `to_f` en la frontera porque la temperatura puede tener decimales; `9.0/5` en lugar de `9/5` para esquivar la division entera ($9/5 = 1$, y el resultado saldria mal); `round(1)` para mostrar un decimal; comentarios que documentan las formulas para quien lea el codigo. Pruebalo con 0 (debe dar 32 F y 273.2 K) y con 100 (212 F y 373.2 K): si esos dos casos salen bien, tu formula es correcta.

Constantes: valores que no deberian cambiar

Las constantes guardan valores fijos que tu programa usa pero que no cambian durante la ejecucion: tasas de impuestos, limites, mensajes estandar, URLs base. Se escriben en MAYUSCULAS_CON_GUIONES y por convencion se definen una vez, al inicio del archivo o del programa:

```
TASA_IVA = 0.16
INTENTOS_MAXIMOS = 3
NOMBRE_ACADEMIA = "Lucas EGA Academy"

precio = 100
total = precio * (1 + TASA_IVA)
p total # 115.99999999999999 (el fantasma del Float, redondea al mostrar)
```

Por que usar constantes en lugar de escribir 0.16 directo en cada formula: primero, si el IVA cambia a 0.18, lo cambias en UN lugar en vez de cazar docenas de 0.16 regados por el codigo

(y rezar por no olvidar uno). Segundo, TASA_IVA se explica sola; 0.16 suelto en una formula obliga al lector a adivinar que es (los programadores llaman a esos numeros magicos "magic numbers" y los odian). Tercero, si alguien intenta reasignar una constante, Ruby emite una advertencia: no lo impide (puedes hacerlo), pero te avisa que estas haciendo algo sospechoso.

```
PI = 3.1416
PI = 3.14 # warning: already initialized constant PI (pero lo permite)
```

Esa advertencia es Ruby diciendote "oye, esto se suponía fijo". En código profesional, reasignar constantes es casi siempre un bug.

Código malo (numeros magicos):

```
total = precio * 1.16
envio = total > 500 ? 0 : 99
```

Que es 1.16? Que es 500? Que es 99? Dentro de seis meses ni el autor lo recuerda. Código bueno:

```
TASA_IVA = 0.16
ENVIO_GRATIS_DESDE = 500
COSTO_ENVIO = 99

total = precio * (1 + TASA_IVA)
envio = total > ENVIO_GRATIS_DESDE ? 0 : COSTO_ENVIO
```

El código bueno se lee como las reglas del negocio. Buena práctica de la industria: toda regla de negocio expresada como número (tasas, límites, umbrales, plazos) vive en una constante con nombre claro, definida arriba del todo. Es una de las refactorizaciones más baratas y con más retorno que existen.

El objeto Integer como navaja suiza: metodos que no esperabas

En la unidad 1 te dije que en Ruby todo es un objeto, y los números no son la excepción. Ya viste even?, odd?, abs, round, ceil, floor. Pero Integer esconde más trucos que los principiantes tardan en descubrir y que los profesionales usan a diario.

times: ejecuta un bloque exactamente n veces. Es el bucle más simple de Ruby y lo veras por todas partes:

```
3.times do
  puts "Hola"
end
# Imprime Hola tres veces
```

upto y downto: cuentan de un numero a otro, pasando el valor actual al bloque:

```
1.upto(5) { |n| puts "voy en #{n}" } # 1, 2, 3, 4, 5
5.downto(1) { |n| puts "faltan #{n}" } # 5, 4, 3, 2, 1
```

chr y ord: convierten entre numeros y caracteres (codigo ASCII/Unicode). Util para cifrados sencillos y manipulacion de texto a bajo nivel:

```
p 65.chr # "A"
p "A".ord # 65
```

digits: descompone un numero en sus digitos (util en validaciones y retos de programacion):

```
p 12345.digits # [5, 4, 3, 2, 1] (al reves, del menos al mas significativo)
```

gcd y lcm: maximo comun divisor y minimo comun multiplo, directo en el lenguaje:

```
p 12.gcd(18) # 6
p 4.lcm(6) # 12
```

Y los predicados zero? y positive?/negative? que hacen el codigo mas expresivo que comparar con == 0:

```
cantidad = 0
puts "vacio" if cantidad.zero? # mas legible que cantidad == 0
```

Analogia: si los numeros en otros lenguajes son ladrillos inertes, en Ruby son ladrillos roboticos que saben contarse solos, ordenarse y responder preguntas. No solo guardan valor: HACEN cosas. Cada vez que escribas un bucle manual para contar, preguntate si un metodo de Integer ya lo hace por ti; casi siempre si.

Error comun: olvidar que times/upto/downto con bloque multilinea usan do...end y con una linea usan llaves {}. Ambas funcionan en ambos casos por sintaxis, pero la convencion es: llaves para una linea, do...end para varias. Mezclarlos al reves funciona pero delata al novato ante ojos experimentados.

Strings multilinea y caracteres especiales: mas alla de una linea

Los programas reales manejan textos largos: mensajes de ayuda, plantillas, reportes. Ruby te da herramientas elegantes para eso sin concatenar docenas de lineas.

Primera herramienta: el string multilinea con comillas dobles abarca saltos de linea reales:

```
poema = "Rosas son rojas,  
el cielo es azul,  
Ruby es hermoso,  
y tu codigo tambien."  
puts poema
```

El string conserva los saltos de linea tal cual los escribiste. Util para textos cortos y fijos.

Segunda herramienta: heredoc, el favorito para bloques grandes de texto (lo veras muchisimo en Rails para plantillas y SQL). Empieza con `<<~IDENTIFICADOR` y termina con el identificador solo en una linea:

```
mensaje = <<~TEXT0  
  Bienvenido a #{NOMBRE_ACADEMIA rescue "la academia"}.  
  Tu curso incluye:  
    - 30 clases en video  
    - Ejercicios practicos  
    - Certificado al finalizar  
TEXT0  
puts mensaje
```

El `~` de `<<~` quita la indentacion comun automaticamente, asi tu codigo se ve ordenado y el texto sale limpio. Dentro del heredoc con comillas implicitas funciona la interpolacion igual que en comillas dobles. (El `rescue` dentro de `#{}` es un truco defensivo: si la constante no existe, usa el texto alternativo; no te preocupes por dominarlo ahora.)

Tercera herramienta: secuencias de escape esenciales mas alla de `\n`. Memoriza estas cinco: `\n` (salto de linea), `\t` (tabulador, util para alinear columnas en reportes de consola), `\\` (una barra invertida literal), `\'` (comilla doble dentro de comillas dobles), `\'` (comilla simple dentro de comillas simples). Ejemplo practico, un mini reporte alineado:

```
puts "Nombre\tEdad\tCiudad"  
puts "Ana\t25\tCDMX"  
puts "Luis\t30\tGuadalajara"
```

El `\t` alinea en columnas como una tabla sencilla. Es el truco mas rapido para reportes legibles en consola sin librerias externas.

Cuarta herramienta: `%w` y `%i` para arreglos de palabras y simbolos sin escribir comillas mil veces (adelanto util de la unidad 4):

```
frutas = %w(manzana pera uva)
p frutas # ["manzana", "pera", "uva"]
estados = %i(activo pendiente archivado)
p estados # [:activo, :pendiente, :archivado]
```

`%w` crea un arreglo de strings separando por espacios; `%i` lo mismo pero de simbolos. Cuando tengas listas fijas de palabras, esto ahorra comillas y comas y se lee limpio.

Error comun: en heredoc, el identificador de cierre debe ir EXACTAMENTE igual (TEXTO vs texto son distintos) y en versiones modernas puede ir indentado solo con `<<~`. Con `<<-` (guion) se permite indentar el cierre pero NO se quita la indentacion del contenido; con `<<` (nada) el cierre debe ir al inicio de la linea. Si tu heredoc "se traga" el resto del archivo, el 99% de las veces el identificador de cierre esta mal escrito. Usa siempre `<<~` y olvidate del problema.

Buena practica: para textos que el usuario final va a leer (mensajes, ayuda, reportes), prefiere heredoc o strings multilinea sobre concatenaciones largas con `+`: el texto queda legible en tu codigo tal como saldra en pantalla, y modificarlo no requiere reequilibrar comillas y pluses.

nil: la nada también es un objeto (y tiene sus reglas)

nil es como Ruby dice "aqui no hay nada". No es cero, no es un string vacio, no es falso: es la ausencia de valor. Y como todo en Ruby, es un objeto: de la clase NilClass, con sus propios metodos. Lo vas a encontrar por todas partes: un hash devuelve nil cuando buscas una clave que no existe, un metodo que no encuentra nada devuelve nil, una variable que declaras sin usar aun conceptualmente vale nil.

```
p nil.class # NilClass
p nil.nil?  # true (pregunta: eres la nada? si)
p nil.to_s  # "" (a texto es vacio)
p nil.to_i  # 0 (a entero es cero)
p nil.to_a  # [] (a arreglo es vacio)
```

nil es "amable" con las conversiones: en lugar de explotar, devuelve el valor neutro de cada tipo. Pero cuidado: nil NO responde a la mayoría de los metodos, y llamarlos lanza NoMethodError, el error mas comun en programas reales:

```
nombre = nil
puts nombre.upcase # NoMethodError: undefined method 'upcase' for nil
```

Traducción: "trataste de poner en mayúsculas algo que no existe". Este error aparece cuando un dato no llegó (el usuario no escribió nada, el archivo no traía esa columna, la API no devolvió ese campo) y tu código asumió que sí. La defensa profesional tiene dos niveles. Nivel 1: valores por defecto con `||` (OR lógico). En Ruby, `||` devuelve el primer valor "verdadero":

```
nombre = nil
nombre = nombre || "Invitado" # si nombre es nil, usa "Invitado"
puts "Hola, #{nombre}" # Hola, Invitado

# Atajo idiomático que verás en todo código Ruby:
ciudad ||= "Desconocida" # si ciudad es nil (o false), asigne esto
```

`||=` se lee "o igual": asigna solo si la variable es nil o false. Es EL patrón para defaults en Ruby y lo verás en configuraciones, parámetros y cachés. Nivel 2: el safe navigation operator `&.`, que evita el `NoMethodError` encadenando con cuidado (lo veremos a fondo más adelante, pero reconócelo desde ya):

```
usuario = nil
p usuario&.nombre # nil, sin explotar (en lugar de NoMethodError)
```

Analogía: nil es una caja vacía con etiqueta. No pesa nada, pero la caja EXISTE y puedes preguntarle cosas básicas ("¿estas vacía?" -> nil?). El error clásico es abrir la caja esperando un regalo y enojarte porque no hay nada: el problema no es la caja, es tu suposición. Programa verificando antes de abrir: nil? y `||` son tus formas de preguntar "¿hay algo ahí?" antes de usarlo.

Buena práctica: cuando un método puede devolver nil, documentalo en un comentario ("devuelve nil si no encuentra el usuario") y quien lo llame debe manejar ese caso. Los bugs de nil más caros de la industria son siempre el mismo: alguien asumió que había dato y no había. En Rails existe incluso el patrón "fail fast" (falla rápido y ruidoso) vs "nil silencioso", decisión de diseño que separa sistemas débiles de sistemas robustos.

Formato de números: mostrar dinero, porcentajes y tablas

Calcular es una cosa; MOSTRAR los números bonito es otra, y en programas reales la presentación importa tanto como el cálculo. Nadie quiere ver "Total: 115.99999999999999" en un ticket. Ruby te da tres niveles de formato.

Nivel 1: round con decimales. Ya lo usaste: total.round(2) deja dos decimales. Simple y suficiente para la mayoría de los casos. Pero OJO: round devuelve un Float, y al interpolarlo Ruby puede mostrar menos decimales de los que quieres (115.5 en lugar de 115.50). Para dinero eso se ve mal.

Nivel 2: sprintf / format para control total. Es el formato clasico estilo C, potente y preciso:

```
precio = 115.5
puts sprintf("%.2f", precio) # $115.50 (siempre 2 decimales)
puts format("%.2f", precio)  # lo mismo (format es alias amable)
puts sprintf("%d unidades", 42) # 42 unidades (%d = entero)
puts sprintf("%-10s|%5d", "Ana", 25) # "Ana      |    25" (columnas alineadas)
```

Los codigos: %.2f = decimal con 2 cifras, %d = entero, %s = texto, %-10s = texto alineado a la izquierda en 10 espacios, %5d = entero alineado a la derecha en 5 espacios. Con esto construyes tickets y reportes alineados como profesional sin librerias externas.

Nivel 3: separadores de miles para legibilidad. Ruby permite guiones bajos dentro de numeros literales (1_000_000 se lee mejor que 1000000 y Ruby los ignora), y para mostrarlos con comas hay tecnicas con gsub:

```
numero = 1_000_000 # el guion bajo es solo visual, vale 1000000
p numero # 1000000
# Formato con comas (truco clasico):
p numero.to_s.reverse.gsub(/(\d{3})(?=\d)/, '\\1,').reverse # "1,000,000"
```

Ese truco usa una expresion regular (unidad 9); por ahora usalo como receta magica para reportes. La leccion importante es la primera linea: escribe 1_000_000 en tu codigo siempre que el numero tenga mas de 4 digitos; tu yo del futuro (y tus revisores) lo agradeceran.

Codigo malo (mostrar crudo):

```
puts "Total: #{total}" # Total: 115.99999999999999 (horrible)
```

Codigo bueno:

```
puts sprintf("Total: $.2f", total) # Total: $116.00 (profesional)
```

Buena practica de la industria: separa CALCULO de PRESENTACION. Calcula con la maxima precision que tengas (Float, o centavos Integer para dinero serio) y formatea SOLO al mostrar. Jamas redondees los valores intermedios de un calculo encadenado: los redondeos se

acumulan y el centavo perdido aparece en la conciliación contable. Redondea al final, al pintar en pantalla. Esta regla, simple como suena, evita una categoría entera de bugs financieros.

Glosario de la unidad 2

Variable local: variable con nombre en minúsculas, visible solo donde se crea.

Variable de instancia (@nombre): pertenece a un objeto; vive mientras el objeto viva.

Variable global (\$nombre): visible en todo el programa; evítala, es fuente de bugs.

Constante (NOMBRE): valor fijo por convención; Ruby advierte si la reasignas.

Integer: número entero de precisión arbitraria (no se desborda).

Float: número decimal de punto flotante; no es exacto para dinero.

División entera: dividir dos Integer trunca el resultado ($7/2 = 3$); convierte a Float para decimales.

Módulo (%): devuelve el residuo de una división; útil para pares/impares y ciclos.

String: texto entre comillas; mutable y lleno de métodos útiles.

Interpolación (#{...}): inserta valores dentro de strings con comillas dobles.

Comillas simples (' '): texto literal, sin interpolación ni escapes.

Comillas dobles (" "): texto inteligente, con interpolación y secuencias como `\n`.

Símbolo (:nombre): etiqueta inmutable y única; ideal para nombres internos y claves.

Heredoc (<<~TEXTO): bloque de texto multilinea limpio, con interpolación.

`.class`: método que devuelve la clase de cualquier objeto.

`is_a?`: pregunta si un objeto pertenece a una clase (respetando herencia).

`respond_to?`: pregunta si un objeto sabe hacer algo (base del duck typing).

`gets`: lee una línea del teclado; siempre devuelve String.

`chomp`: quita el salto de línea final del texto leído.

`strip`: quita espacios en blanco de ambos extremos del string.

`to_i` / `to_f` / `to_s`: conversiones a Integer, Float y String.

`Integer()` / `Float()`: conversiones estrictas; lanzan error si el texto no es válido.

nil: la ausencia de valor; es un objeto de la clase NilClass.

||= : asigna un valor por defecto solo si la variable es nil o false.

sprintf/format: formato preciso de numeros y texto (%.2f, %d, %s).

round(n): redondea un numero a n decimales (para mostrar, no para calcular).

Checklist final: verifica que dominas la unidad 2

1. Declaro variables con nombres claros en snake_case y se los cinco tipos por prefijo: local, @instancia, @@clase, \$global (que evito) y CONSTANTE. No confundo = con ==.
2. Opero con Integer y Float sin caer en la division entera: se que $7/2 = 3$ y que para decimales convierto a Float primero. Se que los Float no son exactos y que el dinero serio va en centavos Integer o BigDecimal.
3. Elijo las comillas correctas: dobles para interpolar #{} y escapes, simples para texto literal. Prefiero la interpolacion sobre la concatenacion con +.
4. Uso los metodos de string esenciales (length, upcase/downcase, strip, split, gsub, include?, reverse) y entiendo la convencion ? (pregunta) y ! (modifica el original).
5. Distingo simbolos de strings: :etiqueta para nombres internos, claves de hash y estados; "texto" para lo que se muestra o cambia. Se que cada :simbolo es un unico objeto en memoria.
6. Uso .class, is_a? y respond_to? para diagnosticar tipos, y mi primer reflejo ante un error de metodo es preguntar p variable.class.
7. Leo input con gets.chomp (y .strip cuando hace falta), sabiendo que SIEMPRE llega como String, y convierto en la frontera con to_i/to_f/to_s. Conozco la diferencia entre to_i (tolerante) e Integer() (estricto).
8. Manejo nil sin panico: lo detecto con nil?, le pongo defaults con ||= y se que llamar metodos sobre nil lanza NoMethodError.
9. Defino constantes en MAYUSCULAS para las reglas del negocio (tasas, limites) en lugar de regar numeros magicos por el codigo.
10. Formateo numeros para mostrarlos: round para lo rapido, sprintf("%.2f") para dinero profesional, y separo el calculo (precision maxima) de la presentacion (formato bonito).
11. Hice los 9 ejercicios de la practica y escribi al menos un programa interactivo propio (tienda, conversor, calculadora) que pida datos, calcule y muestre resultados formateados.

Con esto dominas el vocabulario basico de Ruby. La unidad 3 (condicionales) tomara tus variables y decisiones: if, elsif, unless y case para que tus programas empiecen a TOMAR DECISIONES. Ahi es donde dejan de ser calculadoras y empiezan a parecer inteligentes.

Resumen de la unidad

- Las variables son nombres para valores: se crean con nombre = valor, sin declarar tipos. El tipo lo tiene el VALOR, no la variable (tipado dinamico).
- Tipos de variable por prefijo: local (minuscuala), @instancia, @@clase, \$global (evitala), CONSTANTE (mayuscula, valor fijo).
- Nombres en snake_case para variables y metodos, CamelCase para clases, MAYUSCULAS para constantes. Un buen nombre responde "que contiene" sin leer mas codigo. = asigna, == compara: no los confundas.
- Integer para enteros (sin limite de tamano en Ruby), Float para decimales. Operaciones: + - * / % . **Division de dos enteros = entero truncado (7/2 = 3); para decimales convierte a Float primero.**
- Los Float no son exactos (0.1 + 0.2 = 0.30000000000000004): para dinero usa centavos como Integer o BigDecimal, nunca Float crudo.
- Strings: comillas dobles interpolan #{} y entienden \n; comillas simples son literales. Interpolacion sobre concatenacion, siempre que puedas.
- Metodos de string esenciales: length, upcase/downcase/capitalize, strip, gsub, split, include?, reverse. ? pregunta (true/false), ! modifica el original (version peligrosa).
- Simbolos (:nombre): etiquetas inmutables y unicas, un solo objeto en memoria. Para nombres internos, claves de hash y estados. Strings para texto que se muestra o cambia.
- .class te dice el tipo de cualquier objeto; is_a? respeta herencia; respond_to? pregunta por habilidades (duck typing).
- gets lee una linea del teclado y SIEMPRE devuelve String (con \n incluido). gets.chomp lo limpia. Todo input es texto hasta que tu lo conviertas.
- Conversiones: to_i (a entero, tolerante: "hola".to_i = 0), to_f (a decimal), to_s (a texto). Integer()/Float() son estrictos y lanzan error con texto invalido. Convierte en la frontera, justo donde entra el dato.
- Ritual de depuracion de la unidad: p variable (valor real), p variable.class (tipo real), compara con lo esperado. El 90% de los bugs aqui son "crei que era numero pero era string".

- Tabla mental de la unidad: variable = cajita con etiqueta | snake_case = nombres | Integer/Float = numeros (cuidado con 7/2) | " " interpola, ' ' literal | :simbolo = etiqueta unica | .class = detector | gets.chomp = escuchar limpio | to_i/to_f/to_s = convertir en la frontera.

UNIDAD 3: Tomando decisiones

En esta unidad vas a aprender lo que convierte a un programa tonto en un programa inteligente: la capacidad de tomar decisiones. Hasta ahora tus programas hacen lo mismo siempre, corren de arriba a abajo sin pensar. Pero un programa de verdad necesita reaccionar: si el usuario escribe una contraseña incorrecta, rechazarlo; si una temperatura es alta, encender la alarma; si un numero es par, hacer una cosa y si es impar, hacer otra. Esa capacidad de elegir caminos se llama flujo condicional, y en Ruby es una de las partes mas elegantes del lenguaje.

Ruby fue diseñado por Yukihiro Matsumoto con una idea clara: el código debe ser fácil de leer para los humanos, no solo para la maquina. Por eso los condicionales en Ruby se leen casi como español: "if edad es mayor o igual que 18, entonces puedes entrar, si no, no". Y también tiene formas cortas y elegantes como "unless", que significa "a menos que", algo que casi ningún otro lenguaje tiene de forma tan natural.

En las clases 7, 8 y 9 de este curso vas a dominar los condicionales: el clásico if/else/elsif con su obligatorio end, el refinado unless, los operadores de comparación y los operadores lógicos que combinan condiciones, la idea de "truthiness" (que en Ruby es sorprendentemente simple: solo false y nil son falsos, todo lo demás es verdadero), el poderoso case/when que puede comparar rangos y hasta expresiones regulares, el operador ternario para decisiones en una sola línea, y finalmente los bucles while y until con sus herramientas de control break y next para repetir acciones hasta que una condición se cumpla.

Piensa en esta unidad como aprender a darle criterio a tu código. Un programa sin condicionales es como un mesero que sirve lo mismo a todos sin preguntar. Un programa con condicionales es un buen mesero: pregunta, escucha y sirve lo que cada quien pidió. Vamos paso a paso, con analogías de la vida real, errores típicos que todos cometemos al empezar y muchos ejemplos que puedes correr en tu computadora.

Explicación detallada

El condicional if: la decisión básica

Imaginate que vas a entrar a un antro. El cadenero te pide tu identificación, mira tu edad y decide: si tienes 18 o mas, pasas; si no, te quedas fuera. Eso es exactamente un if. El programa evalúa una condición (una pregunta que se responde con si o no) y según la respuesta ejecuta un bloque de código u otro.

En Ruby se escribe así:

```
edad = 20

if edad >= 18
  puts "Puedes entrar"
else
  puts "No puedes entrar"
end
```

Fíjate en tres cosas. Primero, la condición va después del `if` y no necesita parentesis (aunque puedes ponerlos si quieres, Ruby los hace opcionales porque busca legibilidad). Segundo, el bloque del `else` se ejecuta cuando la condición es falsa. Tercero, y esto es importantísimo: todo `if` en Ruby debe cerrarse con la palabra `end`. Si olvidas el `end`, el programa truena con un error de sintaxis. Es el error número uno de los principiantes.

Cuando tienes más de dos caminos, usas `elsif` (nota que se escribe `elsif`, sin la `e` de en medio: no es `"elseif"`, es `"elsif"`; esto confunde a todos al principio). Es como el cadenero que ahora tiene más categorías:

```
calificacion = 85

if calificacion >= 90
  puts "Excelente, sacaste A"
elsif calificacion >= 80
  puts "Muy bien, sacaste B"
elsif calificacion >= 70
  puts "Pasaste con C"
else
  puts "Reprobaste, a estudiar mas"
end
```

Ruby evalúa las condiciones en orden, de arriba a abajo, y ejecuta el primer bloque cuya condición sea verdadera. En cuanto encuentra uno verdadero, ignora todos los demás. Por eso el orden importa: si pusieras primero `calificacion >= 70`, un 95 entraría ahí y nunca llegaría al bloque del 90. Siempre ordena de lo más específico a lo más general, o de mayor a menor cuando son rangos numéricos.

Analogía de la vida real: un `if/elsif/else` es como un semáforo con varias luces. El programa llega a la intersección, mira cuál luz está prendida y actúa en consecuencia. No puede haber dos luces prendidas al mismo tiempo en el sentido de que solo se ejecuta un camino: el primero que coincida.

Error comun de principiante: usar un solo signo de igual en la condicion.

```
# MAL: esto ASIGNA 18 a edad, no la compara
if edad = 18
  puts "Tienes 18"
end
```

Con un solo `=` le estas diciendo a Ruby "guarda 18 dentro de edad", y como una asignacion siempre se considera verdadera (el valor 18 no es false ni nil), el bloque siempre se ejecuta. Para comparar igualdad se usan DOS signos: `==`. Este error es tan comun que hasta programadores con experiencia lo cometen cuando estan cansados. Si tu if se comporta raro y siempre entra al mismo camino, revisa primero los signos de igual.

Buena practica: cuando la condicion es larga, no tengas miedo de partirla en varias lineas o guardarla en una variable con un nombre descriptivo antes del if. Un if legible vale mas que un if compacto.

```
# Regular: funciona pero cuesta leerlo
if usuario_activo == true && intentos < 3 && cuenta_bloqueada == false
  puts "Acceso permitido"
end

# Mejor: la condicion tiene nombre y se entiende sola
puede_entrar = usuario_activo && intentos < 3 && !cuenta_bloqueada

if puede_entrar
  puts "Acceso permitido"
end
```

unless: la elegancia de decir "a menos que"

Aqui es donde Ruby se pone bonito. En la mayoría de los lenguajes, para decir "haz esto si NO se cumple algo" tienes que escribir `if !condicion`. Ruby te da una palabra especial para eso: unless, que significa "a menos que". Lee este codigo en voz alta:

```
lloviendo = false

unless lloviendo
  puts "Sal a pasear"
end
```

"A menos que este lloviendo, sal a pasear". Se lee como español, no como código. Esa es la filosofía de Ruby: que el código parezca lenguaje humano.

El `unless` también puede llevar `else`, aunque se usa poco porque puede confundir:

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

Ojo con esto: un `unless` con `else` se lee al revés y la mayoría de los programadores Ruby prefieren no usarlo. Si te encuentras escribiendo un `unless` con `else`, casi siempre es más claro voltearlo a un `if` normal. La comunidad Ruby tiene una regla no escrita: `unless` es para condiciones simples y positivas en su forma negada; si se complica, usa `if`.

También existe el `unless` en una sola línea, al final de la instrucción, que es muy común en Ruby:

```
puts "Sal a pasear" unless lloviendo
```

Esto se llama "modificador" y lo verás muchísimo en código Ruby real. Es perfecto para guardias rápidas: "haz esto, a menos que pase aquello". Pero úsalo solo cuando la línea se mantiene corta y legible. Si la línea se alarga, mejor el bloque completo con `end`.

Error común: anidar negaciones hasta que nadie entiende nada.

```
# MAL: doble negacion, hay que pensar demasiado
unless !aprobado
  puts "Felicidades"
end
```

``unless !aprobado`` significa "a menos que no este aprobado", o sea "si esta aprobado". Es un trabalenguas. Escríbelo directo:

```
# BIEN: claro y sin vueltas
if aprobado
  puts "Felicidades"
end
```

Regla de oro: si tienes que leer la línea dos veces para entenderla, reescribela.

Operadores de comparacion: las preguntas que hace el programa

Los condicionales viven de hacer preguntas, y los operadores de comparacion son la forma de preguntar. Todos devuelven true o false. Aqui estan los que vas a usar todos los dias:

```
5 == 5    # true, igualdad: son iguales?
5 != 4    # true, desigualdad: son diferentes?
5 > 4     # true, mayor que
5 < 4     # false, menor que
5 >= 5    # true, mayor o igual que
5 <= 4    # false, menor o igual que
```

En Ruby también puedes comparar cadenas de texto con ==, y funciona comparando letra por letra:

```
"hola" == "hola"    # true
"hola" == "Hola"    # false, las mayusculas y minusculas son diferentes
```

Esto ultimo es una fuente clasica de bugs: el usuario escribe "Si" con mayuscula y tu programa esperaba "si" con minuscula. Mas adelante veras como normalizar texto con downcase, pero por ahora tenlo presente: en Ruby "Hola" y "hola" son cosas distintas.

Hay un operador especial que merece mencion aparte: el operador nave espacial `<=>`. Se llama asi porque parece una navecita. Compara dos valores y devuelve -1 si el primero es menor, 0 si son iguales y 1 si el primero es mayor:

```
5 <=> 3    # 1
3 <=> 5    # -1
5 <=> 5    # 0
```

No lo usaras en un if directamente, pero es la base de como Ruby ordena cosas con sort. Saber que existe te ayudara a entender errores y documentacion mas adelante.

Y otro operador util: `===`, el operador de "igualdad de caso". No lo confundas con ==. Se usa internamente en el case/when que veras mas adelante, y sirve para preguntar "este valor cabe dentro de este patron?":

```
(1..10) === 5    # true, el 5 esta dentro del rango 1 a 10
```

No te preocupes si aun no te queda claro; cuando lleguemos a case/when lo veras en accion y hara clic.

Error comun de principiante: comparar cosas de tipos distintos y esperar que Ruby adivine.

```
# MAL: truena con error, no se puede comparar texto con numero asi
if "20" > 18
  puts "Mayor"
end
```

Ruby es estricto: no compara peras con manzanas automaticamente. Si el dato viene como texto (por ejemplo, de gets que siempre devuelve string), conviértelo primero con `to_i` o `to_f` antes de comparar. Este es EL error mas comun cuando usas gets con if, asi que grabatelo: gets entrega texto, conviértelo a numero antes de comparar con numeros.

Operadores logicos: combinar preguntas

En la vida real las decisiones rara vez dependen de una sola pregunta. "Puedes entrar al antro si tienes 18 o mas Y traes identificacion". "Hay descuento si eres estudiante O eres adulto mayor". Para combinar condiciones, Ruby tiene tres operadores logicos: `&&` (y), `||` (o) y `!` (no).

```
edad = 20
trae_id = true

if edad >= 18 && trae_id
  puts "Puedes entrar"
end
```

El `&&` exige que TODAS las condiciones sean verdaderas. El `||` se conforma con que AL MENOS UNA sea verdadera. El `!` invierte el valor: convierte true en false y false en true.

```
es_estudiante = true
es_adulto_mayor = false

if es_estudiante || es_adulto_mayor
  puts "Tienes descuento"
end

lloviendo = false

if !lloviendo
  puts "Sal a pasear"
end
```


Ruby también tiene las versiones con palabras: and, or y not. Hacen lo mismo pero tienen una prioridad distinta que causa sorpresas desagradables. La recomendación de toda la comunidad es simple: usa siempre &&, || y !, y olvídate de and/or/not. Si ves and en código ajeno, ya sabes que es válido pero no es lo recomendado.

Un detalle importante: && y || usan "evaluación de cortocircuito". Suena elegante pero la idea es simple: si con la primera condición ya se sabe el resultado, Ruby ni siquiera evalúa la segunda. En `false && algo`, como el && necesita que todo sea verdadero y ya encontró un falso, no pierde tiempo evaluando `algo`. En `true || algo`, como el || ya encontró un verdadero, tampoco evalúa lo demás. Esto no solo es más rápido: te permite escribir guardias seguras como esta:

```
if usuario != nil && usuario.admin?  
  puts "Bienvenido, admin"  
end
```

Si usuario es nil, la primera condición es falsa y Ruby jamás intenta llamar a .admin? sobre nil, lo que evitaría un error. Si el orden fuera al revés, el programa tronaría. El orden de las condiciones no es decorativo: pon primero la más barata o la que protege a las demás.

Las tablas de verdad, sin tanto rollo académico:

- `true && true` da true. Cualquier otra combinación con && da false.
- `false || false` da false. Cualquier otra combinación con || da true.
- `!true` da false, `!false` da true.

Error común: confundir && con || al negar.

```
# Quieres decir: "si NO es lunes y NO es martes"  
# MAL: esto significa "si no es lunes o no es martes", que casi siempre es true  
if dia != "lunes" || dia != "martes"  
  puts "No es inicio de semana"  
end
```

Piénsalo: si día es "miercoles", la primera parte es true ("miercoles" != "lunes") y con || basta una verdadera para que todo sea true. Pero si día es "lunes", la segunda parte es true ("lunes" != "martes") y también da true. O sea que este if SIEMPRE es verdadero, hagas lo que hagas. Lo correcto es usar &&:

```
# BIEN: las dos negaciones deben cumplirse a la vez
if dia != "lunes" && dia != "martes"
  puts "No es inicio de semana"
end
```

Este es un error logico, no de sintaxis: el programa corre sin tronar pero hace lo incorrecto, que es peor. Cuando un if con negaciones se comporta raro, dibuja la tabla de verdad en papel. En serio, el papel no falla.

Buena practica: si una condicion con && y || mezclados se vuelve confusa, usa parentesis para dejar clara la intencion, aunque Ruby no los exija:

```
if (es_admin || es_dueno) && cuenta_activa
  puts "Acceso total"
end
```

Los parentesis son como las comas en una frase larga: no cambian las palabras, pero evitan malentendidos.

Truthiness: en Ruby casi todo es verdadero

Aqui viene una de las ideas mas liberadoras de Ruby. En muchos lenguajes, el 0 es falso, el texto vacio es falso, el arreglo vacio es falso. En Ruby NO. En Ruby solo hay DOS valores falsos en todo el universo: false y nil. Todo lo demas, absolutamente todo, se considera verdadero en un condicional. Esto se llama "truthiness" (que tan verdadero es algo).

```
if 0
  puts "El cero es verdadero en Ruby"
end

if ""
  puts "El texto vacio es verdadero en Ruby"
end

if []
  puts "El arreglo vacio es verdadero en Ruby"
end
```

Los tres mensajes se imprimen. Si vienes de JavaScript, Python o C, esto te va a sacar de onda al principio, porque en esos lenguajes el 0 o el "" son falsos. En Ruby, no. Grabatelo como mantra: solo false y nil son falsos.

Esto tiene una consecuencia practica muy bonita: puedes usar directamente el valor devuelto por un metodo como condicion. Por ejemplo, muchos metodos devuelven nil cuando algo sale mal y un valor util cuando sale bien:

```
nombre = gets.chomp

if nombre
  puts "Hola, #{nombre}"
end
```

Pero cuidado: como el texto vacio "" es verdadero en Ruby, si el usuario solo presiona Enter, nombre sera "" y el if lo tomara como verdadero. Si quieres rechazar textos vacios, tienes que preguntarlo explicitamente:

```
nombre = gets.chomp

if nombre != ""
  puts "Hola, #{nombre}"
else
  puts "No escribiste nada"
end
```

O mas elegante, usando el metodo empty? que tienen los strings:

```
if !nombre.empty?
  puts "Hola, #{nombre}"
end
```

El nil merece atencion especial porque aparece todo el tiempo: es lo que Ruby devuelve cuando algo "no existe". Buscar una llave que no esta en un hash devuelve nil. Un metodo que no encuentra lo que buscaba devuelve nil. Por eso veras mucho codigo que pregunta `if resultado` para saber si una busqueda tuvo exito: si es nil, fallo; si trae algo, funciono.

Error comun: asumir que 0 es falso al contar cosas.

```
# MAL si vienes de otro lenguaje: en Ruby el 0 es verdadero
intentos_restantes = 0

if intentos_restantes
  puts "Aun tienes intentos" # Se imprime aunque sean 0!
end
```

Si quieres preguntar por cantidades, compara explícitamente:

```
# BIEN: pregunta lo que realmente quieres saber
if intentos_restantes > 0
  puts "Aun tienes intentos"
else
  puts "Se acabaron los intentos"
end
```

La lección general: en los condicionales pregunta exactamente lo que quieres saber, no confíes en atajos de otros lenguajes. Ruby premia la claridad.

case/when: el semáforo elegante

Cuando tienes muchas opciones basadas en el MISMO valor, encadenar `elsif` se vuelve tedioso. Para eso existe `case/when`. Es como un `if/elsif` especializado en comparar una sola expresión contra varios candidatos:

```
dia = "lunes"

case dia
when "lunes"
  puts "Inicio de semana, ánimo"
when "viernes"
  puts "Ya casi es fin de semana"
when "sabado", "domingo"
  puts "A descansar"
else
  puts "Día normal de trabajo"
end
```

Fíjate que puedes poner varios valores separados por comas en un mismo `when`, como "sabado", "domingo". Y el `else` funciona igual que en el `if`: es el camino por defecto si nada coincide.

Pero el `case/when` en Ruby es mucho más poderoso que el `switch` de otros lenguajes, porque no solo compara con `==`: usa el operador `===` que viste antes, lo que significa que puedes comparar contra RANGOS:

```

calificacion = 85

case calificacion
when 90..100
  puts "Sacaste A"
when 80..89
  puts "Sacaste B"
when 70..79
  puts "Sacaste C"
else
  puts "Reprobaste"
end

```

`90..100` es un rango que incluye todos los numeros del 90 al 100. El when pregunta "este valor cae dentro de este rango?" usando ==. Esto reemplaza de forma hermosa a los elsif con comparaciones de >= y <=, y es mucho mas dificil equivocarse con los limites.

Y aun mas: puedes comparar contra EXPRESIONES REGULARES, que son patrones de texto:

```

email = "juan@example.com"

case email
when /@gmail\.com$/
  puts "Usas Gmail"
when /@hotmail\.com$/
  puts "Usas Hotmail"
else
  puts "Otro proveedor"
end

```

Las expresiones regulares van entre diagonales // y son un tema avanzado, pero que el case las soporte muestra su flexibilidad. También puedes comparar contra clases de objetos (`when String`, `when Integer`), lo que sirve para validar tipos de datos.

Analogia: si el if/elsif es un cadenero que hace una pregunta distinta a cada persona, el case/when es un clasificador de correo: toma UN paquete (la expresion del case) y lo manda al casillero correspondiente (el when que coincida). Un solo valor de entrada, muchos destinos posibles.

Error comun: olvidar que el case también necesita end. Todo bloque que abre con una palabra clave en Ruby (if, unless, case, while, def, do) se cierra con end. Si tu editor no te ayuda con esto, cuenta las aperturas y los cierres: deben coincidir.

Otro error tipico: poner condiciones completas en el when en lugar de valores.

```
# MAL: el when no lleva comparaciones, lleva valores o patrones
case edad
when edad >= 18
  puts "Mayor"
end
```

El when ya esta comparando edad contra lo que pongas; si pones `edad >= 18` ahi, Ruby compara edad contra true/false, que no es lo que quieres. Para condiciones arbitrarias usa if/elsif; para comparar un valor contra candidatos, usa case/when. Cada herramienta en su lugar.

Buena practica: el case/when brilla cuando hay 3 o mas opciones sobre el mismo valor. Con solo 2 opciones, un if/else normal se lee mejor. No uses un martillo para poner un chinche.

El operador ternario: decisiones en una linea

A veces la decision es tan simple que un if completo de 5 lineas se siente exagerado. Para eso existe el operador ternario, llamado asi porque tiene tres partes: condicion ? valor_si_true : valor_si_false.

```
edad = 20
mensaje = edad >= 18 ? "Mayor de edad" : "Menor de edad"
puts mensaje
```

Se lee: "si edad >= 18 entonces 'Mayor de edad', si no 'Menor de edad'". Es perfecto para asignar un valor u otro según una condicion simple.

También sirve directo en un puts o en un return:

```
puts edad >= 18 ? "Puedes entrar" : "No puedes entrar"
```

Pero ojo: el ternario es como el chile en la comida, un poco da sabor, mucho arruina el platillo. En cuanto la condicion se complica o necesitas mas de una linea por camino, regresa al if normal:

```
# MAL: ternario anidado, un acertijo para quien lo lea
mensaje = edad >= 18 ? (tiene_id ? "Pasa" : "Trae tu ID") : "Eres menor"

# BIEN: el if normal no tiene nada de malo
if edad >= 18
  if tiene_id
    mensaje = "Pasa"
  else
    mensaje = "Trae tu ID"
  end
else
  mensaje = "Eres menor"
end
```

Regla practica: si el ternario no cabe comodamente en una linea de unas 80 caracteres o necesitas parens anidados para entenderlo, usa if. La brevedad no sirve si nadie (incluido tu yo del futuro) entiende el codigo.

while: repetir mientras algo sea cierto

Hasta aqui las decisiones eran de una sola vez: el programa elegia un camino y seguia. Pero muchas tareas requieren REPETIR algo hasta que una condicion se cumpla: pedir la contrasena hasta que sea correcta, contar del 1 al 10, procesar filas de un archivo hasta que se acaben. Para eso estan los bucles, y el mas basico es while, que significa "mientras".

```
contador = 1

while contador <= 5
  puts "Contando: #{contador}"
  contador = contador + 1
end
```

Esto imprime los numeros del 1 al 5. El while evalua la condicion ANTES de cada vuelta: si es verdadera, ejecuta el bloque y vuelve a preguntar; si es falsa, sale del bucle y sigue con el resto del programa. No olvides el end, que también cierra los bucles.

La parte critica de todo while es que ALGO dentro del bloque debe acercar la condicion a volverse falsa. En el ejemplo, el contador aumenta en cada vuelta hasta que `contador <= 5` deja de cumplirse. Si olvidas aumentar el contador, la condicion siempre es verdadera y el bucle nunca termina: es el temido bucle infinito, del que hablaremos al final de la unidad.

Analogia: un while es como lavar trastes. La condicion es "mientras haya trastes sucios". Cada vuelta lavas uno (el contador avanza). Cuando ya no hay trastes, paras. Si nunca sacas trastes del fregadero (no actualizas la variable), lavas para siempre.

Un uso clasico del while es validar la entrada del usuario: seguir preguntando hasta que responda algo aceptable:

```
contrasena = ""

while contrasena != "secreto123"
  print "Escribe la contrasena: "
  contrasena = gets.chomp
end

puts "Acceso concedido"
```

El programa no avanza hasta que el usuario escriba la contrasena correcta. Fijate que inicializamos contrasena como "" para que la condicion sea verdadera la primera vez y el bucle arranque. Este patron de "inicializar antes, preguntar dentro" lo usaras muchisimo.

Error comun numero uno: el bucle infinito por olvidar actualizar la variable de control.

```
# MAL: contador nunca cambia, esto corre para siempre
contador = 1

while contador <= 5
  puts "Contando: #{contador}"
end
```

Si esto te pasa (y te va a pasar), no entres en panico: presiona Ctrl + C en la terminal para detener el programa. Todos los programadores hemos creado bucles infinitos; la diferencia entre un novato y un experto es que el experto ya sabe donde esta el Ctrl + C.

Error comun numero dos: el clasico error de "uno mas, uno menos" (off by one). Quieres contar del 1 al 10 pero usas `< 10` en lugar de `<= 10` y te falta el ultimo, o al reves. Cuando un bucle te da un numero equivocado de vueltas, revisa los limites de la condicion con un ejemplo pequeno en papel.

until: repetir hasta que algo sea cierto

El until es el hermano gemelo del while, pero con la logica invertida: repite el bloque MIENTRAS la condicion sea FALSA, y se detiene cuando la condicion se vuelve VERDADERA. Su nombre lo dice: "hasta que".

```
contador = 1

until contador > 5
  puts "Contando: #{contador}"
  contador = contador + 1
end
```

Esto hace exactamente lo mismo que el while de antes, pero la condicion esta planteada en positivo: "hasta que el contador sea mayor que 5". Comparalo:

- while: "mientras el contador sea menor o igual a 5, sigue"
- until: "hasta que el contador sea mayor que 5, sigue"

Cual usar? El que se lea mas natural para lo que quieres expresar. El ejemplo de la contraseña queda hermoso con until:

```
contrasena = ""

until contrasena == "secreto123"
  print "Escribe la contraseña: "
  contrasena = gets.chomp
end

puts "Acceso concedido"
```

"Hasta que la contraseña sea correcta, sigue preguntando". Se lee de maravilla. Esa es la razon de existir del until: evitar la negacion en la condicion. Compara `while contrasena != "secreto123"` con `until contrasena == "secreto123"`: el segundo no tiene el != que obliga a pensar al reves.

Al igual que unless, el until también funciona como modificador en una sola linea, y también como el while:

```
contador = contador + 1 while contador < 10
intentos = intentos + 1 until exito
```

Usalos con moderacion y solo cuando la linea queda corta y clara.

Precaucion: no mezcles negaciones con until, igual que con unless. `until !terminado` es un trabalenguas; mejor `while !terminado` o replantea la variable para que la condicion quede en positivo. La meta siempre es que el codigo se lea de una pasada.

break y next: el control fino dentro del bucle

A veces necesitas salir de un bucle antes de que la condicion se cumpla, o saltarte una vuelta sin salir del todo. Para eso existen break (rompe el bucle y sale) y next (salta a la siguiente vuelta). Son como los frenos y el clutch de un carro estandar: el while/until es el acelerador, pero necesitas control fino.

El break es util cuando encuentras lo que buscabas y ya no tiene sentido seguir dando vueltas:

```
numero_secreto = 7
intentos = 0

while true
  intentos = intentos + 1
  print "Adivina el numero (1-10): "
  intento = gets.chomp.to_i

  if intento == numero_secreto
    puts "Adivinaste en #{intentos} intentos!"
    break
  elsif intento < numero_secreto
    puts "Muy bajo, intenta de nuevo"
  else
    puts "Muy alto, intenta de nuevo"
  end
end
```

Fijate en el `while true`: es un bucle infinito deliberado, y el break es la unica salida. Este patron es super comun en programas interactivos: "repite para siempre hasta que el usuario haga lo correcto o pida salir". Sin break, este patron seria imposible o muy torpe.

El next salta el resto de la vuelta actual y pasa directo a la siguiente. Es perfecto para ignorar casos especiales sin anidar ifs:

```

contador = 0

while contador < 10
  contador = contador + 1

  if contador % 2 == 0
    next # los pares se saltan, no se imprimen
  end

  puts "Impar: #{contador}"
end

```

Esto imprime solo los impares del 1 al 10. Sin next, tendrías que meter el puts dentro de un if/else, anidando un nivel mas. El next aplana el código: "si es un caso que no me interesa, sigue con lo que sigue".

Ojo con un detalle sutil: en el ejemplo, el contador se incrementa ANTES del next. Si lo incrementaras después del next, los números pares causarían un bucle infinito porque el contador nunca avanzaría en esas vueltas. Regla: cuando uses next, asegurate de que la variable de control avance antes del salto, o que el salto no impida su avance.

También puedes usar break con un valor, y el bucle devolverá ese valor. Esto es avanzado pero elegante:

```

resultado = while true
  numero = rand(1..100)
  break numero if numero > 90
end

puts "Salio un numero mayor que 90: #{resultado}"

```

En Ruby TODO es una expresión que devuelve un valor, incluidos los bucles. No necesitas dominar esto ahora, pero cuando lo veas en código ajeno ya sabrás que significa.

Error común: confundir break con next. Recuerda: break ROMPE y sale del bucle por completo; next SALTA solo la vuelta actual y sigue con la siguiente. Si tu bucle termina antes de tiempo cuando solo querías saltar un caso, probablemente pusiste break donde iba next.

Bucles infinitos controlados: el while true con proposito

Ya vimos el `while true` en el juego de adivinanzas. Vamos a formalizar el concepto porque es un patron profesional, no un truco sucio: un bucle infinito CONTROLADO es aquel que esta disenado para correr indefinidamente hasta que ocurra un evento especifico que lo detiene con break. La palabra clave es "controlado": siempre hay una salida clara y alcanzable.

Los menus de programas son el ejemplo perfecto:

```
while true
  puts ""
  puts "=== MENU ==="
  puts "1. Ver saldo"
  puts "2. Depositar"
  puts "3. Retirar"
  puts "4. Salir"
  print "Elige una opcion: "
  opcion = gets.chomp

  case opcion
  when "1"
    puts "Tu saldo es $1,000"
  when "2"
    puts "Funcion de deposito (proximamente)"
  when "3"
    puts "Funcion de retiro (proximamente)"
  when "4"
    puts "Gracias por usar el cajero. Adios!"
    break
  else
    puts "Opcion invalida, intenta de nuevo"
  end
end
```

El programa muestra el menu una y otra vez hasta que el usuario elige salir. Sin el bucle infinito, tendrías que duplicar código o usar recursión torpe. Este patron aparece en cajeros automaticos, juegos, chats y cualquier programa interactivo.

La diferencia entre un bucle infinito controlado y uno accidental es una sola cosa: el break alcanzable. Antes de escribir `while true`, preguntate: "cual es la condicion de salida y el usuario (o el programa) puede alcanzarla siempre?". Si la respuesta es si, adelante. Si no, tienes un bug esperando a pasar.

Estrategia de depuración para bucles: cuando un bucle no termina o termina mal, agrega un puts temporal dentro que muestre la variable de control en cada vuelta. Ver los valores en pantalla revela el problema en segundos:

```
# Depuración temporal: ver que hace el contador
while contador <= 5
  puts "[debug] contador vale: #{contador}"
  contador = contador + 1
end
```

Cuando encuentres el problema, borra o comenta la línea de debug. Los programadores profesionales hacen esto todos los días; no es trampa, es técnica.

Ejemplos paso a paso

Vamos a ver ocho programas completos que usan condicionales y bucles while/until. Cada uno está pensado para que lo copies, lo corras y lo modifiques. La mejor forma de aprender es romper el código a propósito y ver que pasa: cambia un `>=` por un `>`, quita un `end`, pon un `until` donde había un `while`. El error también enseña.

Ejemplo 1: clasificador de edades. Este programa pide la edad y dice en qué etapa de la vida estás. Usa `if/elsif/else` con rangos ordenados de mayor a menor especificidad. Fíjate que primero pregunta por los bebés y va subiendo: si preguntara primero "`edad >= 13`", un adulto de 30 entraría ahí por error.

```
print "Cuantos años tienes? "
edad = gets.chomp.to_i

if edad < 0
  puts "Esa edad no existe, no me mientas"
elsif edad <= 2
  puts "Eres un bebé"
elsif edad <= 12
  puts "Eres un niño"
elsif edad <= 19
  puts "Eres un adolescente"
elsif edad <= 64
  puts "Eres un adulto"
else
  puts "Eres un adulto mayor, respeto total"
end
```

El truco esta en el orden: como Ruby ejecuta el PRIMER bloque verdadero, cada elsif solo se evalua si los anteriores fueron falsos. Cuando llegamos a `edad <= 12`, ya sabemos que edad es mayor que 2, asi que no hace falta escribir `edad > 2 && edad <= 12`. El orden hace el trabajo de los limites inferiores por ti. Prueba cambiar el orden de dos elsif y veras como se rompe la logica: es el mejor ejercicio para entender por que el orden importa.

Ejemplo 2: validacion de contrasena con intentos limitados. Aqui combinamos while con un contador de intentos y break. El programa da 3 oportunidades; si el usuario acierta, sale con break y un mensaje de exito; si agota los intentos, el while termina solo y un if después del bucle detecta que no hubo exito.

```
contrasena_correcta = "ruby2026"
intentos = 0
max_intentos = 3
acceso = false

while intentos < max_intentos
  intentos = intentos + 1
  print "Intento #{intentos} de #{max_intentos}. Escribe la contrasena: "
  intento = gets.chomp

  if intento == contrasena_correcta
    acceso = true
    break
  else
    puts "Contrasena incorrecta"
  end
end

if acceso
  puts "Bienvenido al sistema"
else
  puts "Cuenta bloqueada por seguridad"
end
```

Este ejemplo junta varias ideas de la unidad: el while con contador, el break para salir antes de tiempo, y una variable bandera (acceso) que recuerda lo que paso dentro del bucle para usarlo después. La variable bandera es un patron fundamental: el bucle hace el trabajo sucio y la bandera guarda el resultado. Fijate también en que el mensaje final esta FUERA del bucle, después del end: decide según la bandera. Muchos principiantes meten el mensaje final dentro del bucle y se imprime varias veces.

Ejemplo 3: calculadora de descuentos con case/when y rangos. Una tienda da descuentos según el monto de la compra. Los rangos del case hacen que el código sea limpiísimo comparado con una escalera de elsif.

```
print "Cuanto fue tu compra? $"
monto = gets.chomp.to_f

case monto
when 0..99.99
  descuento = 0
when 100..499.99
  descuento = 5
when 500..999.99
  descuento = 10
else
  descuento = 15
end

ahorro = monto * descuento / 100
total = monto - ahorro

puts "Descuento aplicado: #{descuento}%"
puts "Ahorras: ${ahorro.round(2)}"
puts "Total a pagar: ${total.round(2)}"
```

Nota dos detalles: primero, usamos `to_f` porque el monto puede tener centavos, y `to_i` los cortaría. Segundo, el case asigna el descuento a una variable y el cálculo se hace después, una sola vez. Esto evita repetir la fórmula en cada when. Es un principio general: el condicional decide el DATO (el porcentaje), y el código de después hace el CÁLCULO una sola vez. Así, si mañana cambia la fórmula, la cambias en un solo lugar.

Ejemplo 4: juego de adivinanza con pistas de "frio o caliente". El programa elige un número al azar entre 1 y 100, y el usuario lo adivina con pistas. Usa `while true` con `break`, `if/elsif` para las pistas, y el método `abs` para medir la distancia.

```

secreto = rand(1..100)
intentos = 0

puts "Pense un numero del 1 al 100. Adivinalo!"

while true
  intentos = intentos + 1
  print "Intento #{intentos}: "
  numero = gets.chomp.to_i

  if numero == secreto
    puts "Correcto! Lo adivinaste en #{intentos} intentos"
    break
  end

  distancia = (numero - secreto).abs

  if distancia <= 5
    puts "Hirviendo! Estas muy cerca"
  elsif distancia <= 15
    puts "Caliente, te acercas"
  elsif distancia <= 30
    puts "Tibio..."
  else
    puts "Frio, muy lejos"
  end

  if numero < secreto
    puts "Pista: el numero es MAYOR"
  else
    puts "Pista: el numero es MENOR"
  end
end
end

```

El metodo `rand(1..100)` genera un numero aleatorio en ese rango, y `.abs` convierte la diferencia a valor absoluto para medir distancia sin importar la direccion. Este juego es un excelente proyecto para practicar porque mezcla todo: bucle infinito controlado, `break`, condicionales anidados y entrada del usuario. Como mejora, intenta agregar un limite de 10 intentos: necesitaras un contador y una condicion de salida extra en el `while` o un segundo `break`.

Ejemplo 5: tabla de multiplicar con `while` anidados. Un bucle dentro de otro bucle. El de afuera recorre los numeros del 1 al 10 (la tabla que queremos), y el de adentro recorre los multiplicadores del 1 al 10.


```

numero = 1

while numero <= 5
  puts "Tabla del #{numero}:"
  multiplicador = 1

  while multiplicador <= 10
    resultado = numero * multiplicador
    puts "  #{numero} x #{multiplicador} = #{resultado}"
    multiplicador = multiplicador + 1
  end

  puts ""
  numero = numero + 1
end

```

Lo crucial en los bucles anidados es REINICIAR la variable del bucle interno (multiplicador = 1) cada vez que empieza una vuelta del externo. Si la inicializas solo una vez arriba de todo, la primera tabla funciona pero las siguientes no imprimen nada, porque el multiplicador ya quedo en 11. Este es un error clasico: cuando un bucle anidado "solo funciona la primera vez", revisa donde inicializas la variable interna.

Ejemplo 6: validador de email simple con unless y operadores logicos. El programa pide un email hasta que tenga formato aceptable: debe contener @ y un punto después del @. Usa until para repetir hasta que sea valido y metodos de string para revisar.

```

email = ""

until email.include?("@") && email.include?(".")
  print "Escribe tu email: "
  email = gets.chomp

  unless email.include?("@") && email.include?(".")
    puts "Ese email se ve raro, intenta de nuevo"
  end
end

puts "Email registrado: #{email}"

```

El metodo include? pregunta si un texto contiene otro texto y devuelve true o false. La condicion combina dos preguntas con &&: tiene @ Y tiene punto. El until repite mientras la condicion sea falsa, o sea, mientras el email este incompleto. El unless de adentro evita

mostrar el mensaje de error en la vuelta donde el email ya salio bien (porque el until evalua la condicion al inicio de cada vuelta, y en la ultima vuelta exitosa aun entraria al bloque una vez mas... espera, no: si la condicion ya es verdadera, el until ni siquiera entra. El unless protege el mensaje dentro de las vueltas fallidas). Analiza el flujo vuelta por vuelta con un email malo y uno bueno para ver como interactuan until y unless. Es un gran ejercicio de lectura de codigo.

Ejemplo 7: semaforo con case/when y operador ternario. Programa que simula un semaforo: según el color, dice que hacer. Además usa el ternario para un mensaje rapido de multa.

```
print "Color del semaforo (rojo/amarillo/verde): "  
color = gets.chomp.downcase  
  
case color  
when "rojo"  
  puts "ALTO TOTAL"  
when "amarillo"  
  puts "PRECAUCION, reduce la velocidad"  
when "verde"  
  puts "ADELANTE"  
else  
  puts "Ese color no existe en el semaforo"  
end  
  
multa = color == "rojo" ? "Si te lo pasas, multa segura" : "Vas bien"  
puts multa
```

El downcase convierte lo que escriba el usuario a minusculas para que "Rojo", "ROJO" y "rojo" funcionen igual. Este es el truco estandar para comparar texto que viene del usuario: normalizalo primero. El ternario del final muestra su uso ideal: una decision simple que asigna uno de dos mensajes. Corto, claro, en una linea.

Ejemplo 8: contador de pares e impares con next. El programa revisa los numeros del 1 al 20, salta los multiplos de 3 con next (no los cuenta), y cuenta cuantos pares e impares hay entre los restantes. Usa el operador modulo % que da el residuo de la division.

```

numero = 0
pares = 0
impares = 0

while numero < 20
  numero = numero + 1

  if numero % 3 == 0
    next # los multiplos de 3 no se cuentan
  end

  if numero % 2 == 0
    pares = pares + 1
  else
    impares = impares + 1
  end
end

puts "Pares (sin multiplos de 3): #{pares}"
puts "Impares (sin multiplos de 3): #{impares}"

```

El operador % es tu mejor amigo para preguntas de "es multiplo de" o "es par/impar": `numero % 2 == 0` significa "el residuo de dividir entre 2 es cero", o sea, es par. El next hace que los multiplos de 3 se salten el conteo sin necesidad de anidar todo en un if gigante. Verifica el resultado a mano con los primeros numeros: 1 (impar), 2 (par), 3 (saltado), 4 (par), 5 (impar), 6 (saltado)... y confirma que los totales cuadran. Probar con numeros pequenos y verificar a mano es la forma mas rapida de ganar confianza en tu logica.

Practica

Resuelve cada ejercicio por tu cuenta antes de mirar la solucion. La tentacion de ver la respuesta rapido es grande, pero el aprendizaje esta en el intento, no en la lectura. Si te atorras mas de 15 minutos, mira la solucion, entiendela y luego cierrala y escribela tu solo desde cero.

Ejercicio 1: Escribe un programa que pida un numero y diga si es positivo, negativo o cero. Parece trivial, pero practica la estructura if/elsif/else y el orden de las condiciones.

Solucion explicada:

```

print "Escribe un numero: "
numero = gets.chomp.to_i

if numero > 0
  puts "Es positivo"
elsif numero < 0
  puts "Es negativo"
else
  puts "Es cero"
end

```

Funciona porque las tres opciones son excluyentes y cubren todos los casos: si no es mayor que cero ni menor que cero, forzosamente es cero, así que el `else` lo captura sin necesidad de escribir ``elsif numero == 0``. Aprovechar el `else` para el caso restante es más limpio que enumerar todas las condiciones. El `to_i` es obligatorio porque `gets` entrega texto y no se puede comparar texto con número usando `>` y `<`.

Ejercicio 2: Pide la hora del día (0 a 23) y muestra un saludo: "Buenos días" (6-11), "Buenas tardes" (12-19), "Buenas noches" (20-23 y 0-5). Si el número está fuera de 0-23, muestra "Hora inválida".

Solución explicada:

```

print "Que hora es (0-23)? "
hora = gets.chomp.to_i

case hora
when 6..11
  puts "Buenos días"
when 12..19
  puts "Buenas tardes"
when 20..23, 0..5
  puts "Buenas noches"
else
  puts "Hora inválida"
end

```

El case/when con rangos es la herramienta perfecta aqui porque cada saludo corresponde a un intervalo. Fijate en el when 20..23, 0..5: un solo when puede llevar varios rangos separados por comas, lo que resuelve elegantemente el problema de la noche que se parte en dos pedazos (antes y después de la medianoche). Con if/elsif habrias necesitado `hora >= 20 || hora <= 5`, que funciona pero se lee peor. El else captura cualquier numero fuera de 0-23, como 25 o -3.

Ejercicio 3: Usa unless para mostrar "Puedes votar" solo si la persona tiene 18 o mas. Luego reescribelo con if para comparar cual se lee mejor.

Solucion explicada:

```
print "Tu edad: "  
edad = gets.chomp.to_i  
  
unless edad < 18  
  puts "Puedes votar"  
end
```

Esta version funciona pero tiene una negacion (`edad < 18`) dentro del unless, que es justo lo que la guia de estilo desaconseja: "a menos que tu edad sea menor que 18" obliga a pensar al reves. La version con if es mas directa:

```
if edad >= 18  
  puts "Puedes votar"  
end
```

La leccion: unless brilla cuando la condicion queda en positivo, como `unless lloviendo` o `unless registrado`. Si para usar unless tienes que negar la condicion, mejor usa if. La herramienta correcta depende de como quede la frase, no de usar la palabra de moda.

Ejercicio 4: Pide dos numeros y un operador (+, -, *, /). Usa case/when para hacer la operacion y mostrar el resultado. Si el operador no es valido, muestra un mensaje de error. Si es division entre cero, muestra "No se puede dividir entre cero".

Solucion explicada:

```

print "Primer numero: "
a = gets.chomp.to_f
print "Segundo numero: "
b = gets.chomp.to_f
print "Operador (+, -, *, /): "
op = gets.chomp

case op
when "+"
  puts "Resultado: #{a + b}"
when "-"
  puts "Resultado: #{a - b}"
when "*"
  puts "Resultado: #{a * b}"
when "/"
  if b == 0
    puts "No se puede dividir entre cero"
  else
    puts "Resultado: #{a / b}"
  end
else
  puts "Operador no valido"
end

```

El case compara el operador (un string) contra cuatro candidatos, y el else maneja cualquier otra cosa que escriba el usuario. La division necesita un if anidado porque tiene una condicion extra: el divisor no puede ser cero. Usamos to_f para que la division de 5 / 2 de 2.5 y no 2 (con enteros, Ruby trunca los decimales). Este programa es una mini calculadora real: guárdalo, porque en unidades siguientes le agregaras funciones.

Ejercicio 5: Escribe un programa que pida un numero del 1 al 7 y muestre el dia de la semana con case/when. Luego agrega el operador ternario para mostrar "(fin de semana)" o "(entre semana)" según corresponda.

Solucion explicada:

```

print "Numero del dia (1-7): "
dia = gets.chomp.to_i

nombre = case dia
when 1 then "Lunes"
when 2 then "Martes"
when 3 then "Miercoles"
when 4 then "Jueves"
when 5 then "Viernes"
when 6 then "Sabado"
when 7 then "Domingo"
else "Dia invalido"
end

puts nombre

if dia >= 1 && dia <= 7
  tipo = (dia == 6 || dia == 7) ? "fin de semana" : "entre semana"
  puts "Es #{tipo}"
end

```

Aquí el case se usa como EXPRESION: su resultado se guarda en la variable nombre. En Ruby el case devuelve el valor del when que coincidio, así que puedes asignarlo directamente. El `then` permite escribir cada when en una sola línea cuando el cuerpo es corto. El ternario decide entre dos textos según una condición simple, que es su uso ideal. El if exterior evita mostrar el tipo de día cuando el número fue inválido. Fíjate como cada pieza de la unidad encaja con las demás: case para clasificar, ternario para el detalle rápido, if como guardia.

Ejercicio 6: Con un bucle while, imprime los números pares del 2 al 20. Luego hazlo otra vez pero saltando los múltiplos de 4 usando next. Compara ambas versiones.

Solución explicada:

```

# Version 1: todos los pares del 2 al 20
numero = 2

while numero <= 20
  puts numero
  numero = numero + 2
end

```

En la versión 1, en lugar de revisar cada número y preguntar si es par, el contador avanza de 2 en 2 desde el inicio. Es más eficiente y más simple: si sabes el patrón, genera directamente

los valores que quieres en lugar de filtrar. Esta es una lección general de bucles: a veces es mejor construir la secuencia correcta que filtrar una secuencia mas grande.

```
# Version 2: pares del 2 al 20, saltando multiplos de 4
numero = 2

while numero <= 20
  if numero % 4 == 0
    numero = numero + 2
    next
  end

  puts numero
  numero = numero + 2
end
```

En la version 2 el next salta la impresion cuando el numero es multiplo de 4. Fijate que el contador se incrementa ANTES del next: si no lo hicieramos, el bucle se quedaria atorado en el primer multiplo de 4 para siempre. El resultado impreso es 2, 6, 10, 14, 18. Verifica: los pares del 2 al 20 son 2, 4, 6, 8, 10, 12, 14, 16, 18, 20; quitando multiplos de 4 (4, 8, 12, 16, 20) quedan 2, 6, 10, 14, 18. Cuadra.

Ejercicio 7: Pide al usuario que escriba la palabra "ruby" para continuar. Usa until y cuenta cuantos intentos le tomo. Si lo logra al primer intento, felicitalo de forma especial.

Solucion explicada:

```
intentos = 0
palabra = ""

until palabra == "ruby"
  intentos = intentos + 1
  print "Escribe 'ruby' para continuar: "
  palabra = gets.chomp.downcase
end

if intentos == 1
  puts "Increible, a la primera!"
else
  puts "Listo, te tomo #{intentos} intentos"
end
```


El until repite mientras la palabra sea distinta de "ruby", o sea, hasta que el usuario la escriba bien. El downcase hace que "Ruby", "RUBY" y "ruby" funcionen igual, porque seria frustrante fallar solo por las mayusculas. El contador de intentos se incrementa al inicio de cada vuelta, asi que al salir del bucle tiene el numero exacto de intentos. El if/else final usa el operador ternario... no, mejor dicho, usa un if normal porque los mensajes son largos y distintos; el ternario se reserva para decisiones cortas. Cada herramienta en su lugar, como siempre.

Ejercicio 8: Escribe un programa que simule un cajero: muestra un menu con while true, opciones de ver saldo, depositar y retirar, y salir con break. El saldo inicial es 1000. No permitas retirar mas del saldo disponible (usa un if con mensaje de error).

Solucion explicada:

```

saldo = 1000

while true
  puts ""
  puts "--- CAJERO ---"
  puts "1. Ver saldo"
  puts "2. Depositar"
  puts "3. Retirar"
  puts "4. Salir"
  print "Opcion: "
  opcion = gets.chomp

  case opcion
  when "1"
    puts "Tu saldo es: ${saldo}"
  when "2"
    print "Cuanto depositas? $"
    monto = gets.chomp.to_f
    if monto > 0
      saldo = saldo + monto
      puts "Deposito exitoso. Nuevo saldo: ${saldo}"
    else
      puts "El monto debe ser positivo"
    end
  when "3"
    print "Cuanto retiras? $"
    monto = gets.chomp.to_f
    if monto <= 0
      puts "El monto debe ser positivo"
    elsif monto > saldo
      puts "Fondos insuficientes. Tu saldo es ${saldo}"
    else
      saldo = saldo - monto
      puts "Retiro exitoso. Nuevo saldo: ${saldo}"
    end
  when "4"
    puts "Gracias por su visita"
    break
  else
    puts "Opcion invalida"
  end
end
end

```

Este ejercicio integra toda la unidad: while true con break para el menu, case/when para las opciones, if/elsif para validar montos, y una variable (saldo) que persiste entre vueltas porque esta declarada ANTES del bucle. Ese ultimo punto es clave: si declararas saldo dentro del bucle, se reiniciaria a 1000 en cada vuelta y los depositos se perderian. Las variables que deben sobrevivir entre vueltas van fuera; las temporales de cada vuelta van dentro. Prueba romperlo a proposito: mueve saldo dentro del bucle y observa como los depositos "desaparecen". Entender por que pasa eso vale mas que diez lecturas.

Ejercicio 9: Escribe un programa que pida la edad y diga si la persona puede tramitar su INE (18 o mas), usando unless de forma elegante (condicion en positivo, sin negaciones). Luego muestra con un if equivalente por que el unless quedo mejor o peor.

Solucion explicada:

```
print "Tu edad: "
edad = gets.chomp.to_i

# Version unless: la condicion queda en positivo
unless edad >= 18
  puts "Aun no puedes tramitar tu INE"
else
  puts "Ya puedes tramitar tu INE"
end
```

Hmm, esta version usa unless con else, que la guia de estilo desaconseja porque se lee al reves. La forma elegante de verdad es:

```
if edad >= 18
  puts "Ya puedes tramitar tu INE"
else
  puts "Aun no puedes tramitar tu INE"
end
```

La leccion es honesta: no todo problema merece un unless. El unless brilla en guardias simples sin else ("haz esto a menos que pase aquello"), no en decisiones de dos caminos. Forzar la herramienta elegante donde no encaja produce codigo peor que el simple. Un buen programador no es el que usa las herramientas fancy, sino el que elige la herramienta correcta para cada caso.

Ejercicio 10: Programa "semáforo inteligente": pide el color del semaforo y si hay un peaton cruzando (s/n). Si el color es verde Y no hay peaton, muestra "Avanza". Si el color es verde PERO hay peaton, muestra "Espera al peaton". Si es amarillo muestra "Precaucion: reduce velocidad". Si es rojo muestra "Alto total". Usa case/when combinado con if.

Solucion explicada:

```
print "Color (verde/amarillo/rojo): "  
color = gets.chomp.downcase  
print "Hay peaton cruzando? (s/n): "  
peaton = gets.chomp.downcase == "s"  
  
case color  
when "verde"  
  if peaton  
    puts "Espera al peaton"  
  else  
    puts "Avanza"  
  end  
when "amarillo"  
  puts "Precaucion: reduce velocidad"  
when "rojo"  
  puts "Alto total"  
else  
  puts "Color no valido"  
end
```

El case clasifica por color (un solo valor, varios candidatos: su caso de uso ideal) y dentro del when "verde" un if maneja la segunda dimension (el peaton). Esta combinacion de case + if anidado aparece constantemente en programas reales: el case elige la categoria general y el if afina el detalle. La variable peaton se calcula una sola vez como booleano (gets.chomp.downcase == "s"), asi el if la usa directo sin repetir la comparacion. Precalcular booleanos con nombre antes de los condicionales es un habito que hace el codigo mucho mas legible.

Integrandolo todo: el juego de piedra, papel o tijera

Vamos a cerrar la explicacion con un programa que usa casi todo lo de la unidad: un juego completo de piedra, papel o tijera contra la computadora, con marcador y opcion de seguir jugando. Leelo despacio y detecta cada concepto: while, case, if/elsif, operadores logicos, break.

```

victorias = 0
derrotas = 0

while true
  puts ""
  puts "=== PIEDRA, PAPEL O TIJERA ==="
  puts "Marcador: tu #{victorias} - #{derrotas} computadora"
  print "Elige (piedra/papel/tijera) o 'salir': "
  jugador = gets.chomp.downcase

  break if jugador == "salir"

  unless ["piedra", "papel", "tijera"].include?(jugador)
    puts "Jugada invalida, intenta de nuevo"
    next
  end

  computadora = ["piedra", "papel", "tijera"].sample
  puts "La computadora eligio: #{computadora}"

  if jugador == computadora
    puts "Empate!"
  elsif (jugador == "piedra" && computadora == "tijera") ||
        (jugador == "papel" && computadora == "piedra") ||
        (jugador == "tijera" && computadora == "papel")
    puts "Ganaste esta ronda!"
    victorias = victorias + 1
  else
    puts "Perdiste esta ronda"
    derrotas = derrotas + 1
  end
end

puts ""
puts "Juego terminado. Marcador final: tu #{victorias} - #{derrotas} computadora"

if victorias > derrotas
  puts "Campeon!"
elsif derrotas > victorias
  puts "La computadora gana esta vez"
else
  puts "Empate total"
end

```

Detecta las piezas: el `while true` con `break` para el menu infinito; el `unless` con `include?` como guardia que rechaza jugadas invalidas (aqui el `unless` SI queda elegante: condicion positiva y simple); el `next` que salta la ronda sin contar punto; el `sample` que elige la jugada de la computadora al azar; el `if/elsif/else` que decide el ganador con una condicion compuesta de tres casos unidos por `||` (las tres formas de ganar); y el marcador que persiste porque las variables estan fuera del bucle.

La condicion de victoria merece un segundo vistazo: enumera las tres combinaciones ganadoras (piedra rompe tijera, papel envuelve piedra, tijera corta papel). Si ninguna se cumple y no fue empate, la unica posibilidad restante es la derrota, asi que el `else` la captura sin necesidad de escribir las tres combinaciones perdedoras. Aprovechar el `else` para "todo lo demas" no solo ahorra codigo: elimina la posibilidad de olvidar un caso.

Como retos: agrega la opcion "lagarto" y "spock" (son 5 jugadas, cada una le gana a dos), limita el juego a 5 rondas con un contador y declara al campeon automaticamente, o guarda el historial de jugadas en un arreglo (adelanto de la unidad 4) y muestralo al final. Este juego es el proyecto perfecto para practicar porque es divertido de probar: cada cambio lo verificas jugando.

Resumen de la unidad

- El `if` evalua una condicion y ejecuta un bloque si es verdadera; `elsif` agrega mas caminos y `else` es el camino por defecto. Todo se cierra con `end`, sin excepcion.
- En las comparaciones usa `==` (dos signos); un solo `=` asigna en lugar de comparar y es el error mas comun del universo.
- `unless` significa "a menos que" y reemplaza elegantemente al `if` con negacion, pero solo cuando la condicion queda en positivo y simple. Si necesitas negar para usar `unless`, mejor usa `if`.
- Los operadores logicos `&&` (y), `||` (o) y `!` (no) combinan condiciones. Pon primero la condicion que protege a las demas, gracias a la evaluacion de cortocircuito.
- En Ruby solo `false` y `nil` son falsos; todo lo demas (incluidos `0`, `""` y `[]`) es verdadero. Pregunta explicitamente lo que quieras saber en lugar de confiar en atajos de otros lenguajes.
- `case/when` compara un solo valor contra varios candidatos usando `===`, lo que permite rangos (1..10), expresiones regulares y clases. Es la herramienta ideal cuando hay 3 o mas opciones sobre el mismo valor.

- El operador ternario (condicion ? a : b) sirve para decisiones simples en una linea. Si se anida o se alarga, regresa al if.
- while repite mientras la condicion sea verdadera; until repite hasta que sea verdadera. Algo dentro del bloque debe acercar el fin del bucle o tendras un bucle infinito (Ctrl + C para escapar).
- break sale del bucle por completo; next salta solo la vuelta actual. Con while true + break construyes menus y programas interactivos profesionales.
- gets siempre entrega texto: conviértelo con to_i o to_f antes de comparar con numeros, y usa downcase antes de comparar texto del usuario.

UNIDAD 4: Bucles y arreglos

En la unidad anterior aprendiste a tomar decisiones con condicionales y a repetir acciones con `while` y `until`. Ahora vas a conocer las herramientas que hacen famoso a Ruby: sus iteradores elegantes y sus arreglos. Si el `while` es un martillo confiable, los iteradores de Ruby son una caja de herramientas de precision: `times`, `upto`, `downto` y sobre todo `each`, que es probablemente el metodo mas usado en todo el lenguaje.

La segunda mitad de la unidad es igual de importante: los arreglos (`arrays`). Un arreglo es una coleccion ordenada de valores guardados en una sola variable: una lista de nombres, las calificaciones de un alumno, los productos de un carrito de compras. Casi ningun programa real trabaja con un solo dato aislado; casi todos trabajan con listas. Dominar los arreglos (crearlos, acceder a sus elementos, modificarlos y transformarlos con metodos como `map` y `select`) es lo que separa a quien "sabe algo de Ruby" de quien "programa en Ruby de verdad".

Las clases 10, 11 y 12 de este curso cubren exactamente eso: primero los iteradores numericos y `each`, luego los arreglos desde cero (indices positivos y negativos, rangos en indices), y finalmente los metodos de arreglos mas poderosos: `push`, `pop`, `shift`, `unshift` para modificar; `map/collect` y `select/reject` para transformar y filtrar; `each_with_index` para iterar con posicion; `length/size/empty?` para medir; arreglos anidados para datos en tablas; y `split` y `chars` para convertir texto en arreglos.

Piensa en esta unidad como aprender a manejar multitudes. Un condicional maneja un dato; un arreglo con un iterador maneja cientos o miles con el mismo esfuerzo. Un programa que saluda a una persona es un juguete; un programa que saluda a mil personas de una lista es una herramienta. Vamos a construir esa herramienta paso a paso.

Explicacion detallada

times, upto y downto: los contadores elegantes

En la unidad 3 contabas con `while` y una variable contador que tenias que crear, inicializar, comparar e incrementar a mano. Funciona, pero es mucho tramite para algo tan simple como "haz esto 5 veces". Ruby te da iteradores que hacen todo ese tramite por ti.

El mas simple es `times`: "haz esto N veces".


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

Esto imprime "Hola" 5 veces. Sin contador, sin condicion, sin incremento, sin riesgo de bucle infinito. El do...end delimita el bloque de codigo que se repite. Es el reemplazo natural del while cuando sabes de antemano cuantas vueltas quieres.

Pero times puede darte mas: si le pones una variable entre barras verticales |i|, esa variable toma los valores 0, 1, 2, 3, 4 en cada vuelta (empieza en cero, como casi todo en programacion):

```
5.times do |i|
  puts "Vuelta numero #{i}"
end
```

Esto imprime "Vuelta numero 0" hasta "Vuelta numero 4". Si quieres empezar en 1, suma uno dentro del bloque o usa upto, que es el siguiente iterador.

El metodo upto cuenta desde un numero hasta otro, incluyendo ambos extremos:

```
1.upto(5) do |i|
  puts "Contando: #{i}"
end
```

Imprime del 1 al 5. Y downto hace lo mismo pero hacia abajo:

```
5.downto(1) do |i|
  puts "Despegando en #{i}..."
end

puts "Despegue!"
```

Imprime 5, 4, 3, 2, 1 y luego "Despegue!". Perfecto para cuentas regresivas.

Analogia: si el while es contar con los dedos (tu llevas la cuenta mentalmente), times/upto/downto es usar una calculadora con memoria: le dices el rango y ella se encarga de los detalles. Menos piezas moviles, menos cosas que pueden fallar.

Comparalos con el while equivalente para apreciar la diferencia:

```
# Con while: 4 lineas de tramite por 1 de trabajo
contador = 1
while contador <= 5
  puts "Contando: #{contador}"
  contador = contador + 1
end

# Con upto: cero tramite, solo trabajo
1.upto(5) do |i|
  puts "Contando: #{i}"
end
```

El while no desaparece: lo sigues necesitando cuando NO sabes cuantas vueltas seran (como esperar la contrasena correcta). Pero cuando el numero de vueltas se conoce de antemano, los iteradores son mas cortos, mas claros y a prueba de bucles infinitos. Regla practica: vueltas conocidas, iterador; vueltas desconocidas, while/until.

Error comun: pensar que times empieza en 1.

```
# Si esperabas 1, 2, 3, 4, 5, te llevas una sorpresa: imprime 0, 1, 2, 3, 4
5.times do |i|
  puts i
end
```

El times empieza en 0 porque esta pensado para usarse con indices de arreglos, que también empiezan en 0 (lo veras en la siguiente seccion). Si necesitas empezar en 1, usa `1.upto(5)` o suma 1 dentro del bloque: `puts i + 1`. No es un bug, es una convencion; pero tienes que conocerla para no llevarte sorpresas.

Otro detalle: los bloques también se pueden escribir con llaves {} en lugar de do...end cuando son de una sola linea:

```
5.times { |i| puts "Vuelta #{i}" }
```

Ambas formas son validas. La convencion de la comunidad es: llaves para bloques de una linea, do...end para bloques de varias lineas. No es una regla del lenguaje, es una regla de estilo, pero seguirla hace que tu codigo se vea profesional.

each: el iterador estrella de Ruby

Si tuvieras que quedarte con un solo metodo de todo Ruby, seria each. El each recorre una coleccion elemento por elemento y ejecuta el bloque una vez por cada elemento. Es la forma idiomática (la forma "correcta según la cultura Ruby") de recorrer listas.

```
frutas = ["manzana", "platano", "naranja"]

frutas.each do |fruta|
  puts "Me gusta la #{fruta}"
end
```

En cada vuelta, la variable |fruta| toma el siguiente elemento del arreglo: primero "manzana", luego "platano", luego "naranja". Tu no manejas indices, no calculas el tamaño, no te preocupas por pasarte del final: each lo hace todo.

Compara con como se haria con while e indices (la forma vieja, estilo C):

```
# Forma vieja: funciona, pero es puro tramite
frutas = ["manzana", "platano", "naranja"]
i = 0

while i < frutas.length
  puts "Me gusta la #{frutas[i]}"
  i = i + 1
end
```

La version con each es mas corta, mas clara y no tiene el riesgo de equivocarte en los limites. Por eso en Ruby casi nadie recorre arreglos con while: each es el camino.

El each también funciona con rangos y con hashes (los veras en otra unidad):

```
(1..5).each do |n|
  puts "Numero: #{n}"
end
```

Analogía: each es como un maestro de ceremonias que presenta a los invitados uno por uno. Tu solo dices que hacer con cada invitado ("saludalo", "dale un regalo"); el maestro se encarga de llamarlos en orden y de saber cuando se acabo la lista. Con while, tu eras el maestro de ceremonias Y el que saludaba: doble trabajo, doble posibilidad de error.

Un punto importante: `each` devuelve el arreglo original, no los resultados del bloque. Esto significa que `each` sirve para HACER algo con cada elemento (imprimir, guardar en base de datos, enviar un correo), pero no para TRANSFORMAR el arreglo. Para transformar existe `map`, que veras mas adelante. Confundir `each` con `map` es un error clasico: si quieres un arreglo nuevo con los elementos modificados, `each` no te lo da.

```
numeros = [1, 2, 3]
resultado = numeros.each do |n|
  n * 10
end

puts resultado.inspect # [1, 2, 3], el original, no [10, 20, 30]
```

El bloque calculo `n * 10` pero ese valor se desperdicio porque `each` ignora lo que devuelve el bloque. Para eso necesitas `map`. Lo veremos con detalle en su seccion.

for vs each: el duelo que `each` gana siempre

Ruby tiene un bucle `for` parecido al de otros lenguajes:

```
for fruta in ["manzana", "platano", "naranja"]
  puts "Me gusta la #{fruta}"
end
```

Funciona igual que el `each` del ejemplo anterior. Entonces, cual usar? La respuesta de la comunidad Ruby es unanime: usa `each`, casi siempre. Hay dos razones, una practica y una tecnica.

La razon practica es consistencia: `each` funciona igual en arreglos, rangos, hashes y cualquier coleccion; `for` es un invitado raro que casi nadie usa en codigo Ruby real. Si lees codigo de otros (y lo haras mucho), veras `each` por todos lados y `for` casi nunca.

La razon tecnica es el alcance de las variables. Las variables creadas dentro de un `for` "se escapan" y siguen existiendo después del bucle; las de un bloque `each/do` quedan atrapadas dentro del bloque:

```
for i in [1, 2, 3]
  # ...
end
puts i # funciona, i vale 3: se escapo del for

[1, 2, 3].each do |j|
  # ...
end
puts j # ERROR: j no existe fuera del bloque
```

Que las variables del bloque no se escapen es una VENTAJA: evita que una variable temporal del bucle contamine el resto de tu programa y cause bugs misteriosos mas adelante. Es como la diferencia entre usar un plato desechable (each: lo tiras al terminar) y uno de tu vajilla (for: hay que lavarlo y guardarlo, y si lo dejas sucio estorba).

En resumen: conoce que for existe para cuando lo veas en codigo viejo o en tutoriales de otros lenguajes, pero escribe each. Es lo idiomático, es lo seguro y es lo que esperara cualquier programador Ruby que lea tu código.

Que es un arreglo: tu primera coleccion

Hasta ahora tus variables guardaban UN valor: un numero, un texto. Pero la vida real viene en listas: los alumnos de un salon, los precios de un menu, los mensajes de un chat. Un arreglo (array) es una variable que guarda VARIOS valores ordenados, uno tras otro, como los vagones de un tren.

Se crea con corchetes y los elementos se separan con comas:

```
frutas = ["manzana", "platano", "naranja"]
numeros = [1, 2, 3, 4, 5]
mezcla = ["hola", 42, true, 3.14]
vacio = []
```

En Ruby los arreglos pueden mezclar tipos sin problema: texto, numeros, booleanos, todo junto. Otros lenguajes son mas estrictos, pero Ruby es flexible. Eso no significa que debas mezclar por deporte: un arreglo de cosas del mismo tipo es mas facil de procesar. Pero la flexibilidad esta ahi cuando la necesites.

Cada elemento tiene una POSICION llamada indice, y aqui viene la regla de oro numero uno de los arreglos: los indices EMPIEZAN EN CERO. El primer elemento esta en la posicion 0, el segundo en la 1, el tercero en la 2, y así. Es la convencion mas anti-intuitiva para los

principiantes y la causa de la mitad de los errores con arreglos, así que repetimos: el primero es el 0.

```
frutas = ["manzana", "platano", "naranja"]  
# indice:      0          1          2
```

Para acceder a un elemento, pones su índice entre corchetes después del nombre:

```
puts frutas[0] # "manzana"  
puts frutas[1] # "platano"  
puts frutas[2] # "naranja"
```

Analogía: un arreglo es como un edificio de departamentos. El arreglo es el edificio, cada elemento es un departamento y el índice es el número de puerta. Para visitar a alguien necesitas el número correcto. Y en este edificio raro, el primer departamento es el número 0. Nadie sabe por qué los programadores contamos desde cero (hay razones históricas), pero todos lo hacemos y tienes que acostumbrarte.

También puedes crear arreglos con el método `new` o con atajos especiales:

```
a = Array.new(3)          # [nil, nil, nil], tres espacios vacíos  
b = Array.new(3, 0)       # [0, 0, 0], tres ceros  
c = %w(manzana platano naranja) # atajo para arreglo de strings
```

El atajo `%w` crea un arreglo de strings sin tener que escribir comillas y comas: ``%w(hola mundo)`` es lo mismo que ``["hola", "mundo"]``. Lo verás mucho en código real porque ahorra escritura. Hay un primo `%i` para símbolos, que verás cuando estudiemos símbolos.

Error común: pedir un índice que no existe.

```
frutas = ["manzana", "platano", "naranja"]  
puts frutas[5] # nil, no truena, pero no hay nada ahí
```

En Ruby, pedir un índice fuera del arreglo NO lanza un error: devuelve `nil`, el valor que significa "nada". Esto es amable pero peligroso: tu programa sigue corriendo con un `nil` inesperado y el error aparece más adelante, lejos de la causa. Si un valor te sale `nil` sin explicación, revisa los índices: probablemente pediste una posición que no existe o contaste desde 1 en lugar de 0.

Indices positivos y negativos: leer por los dos extremos

Ya sabes que los indices positivos cuentan desde el inicio empezando en 0. Pero Ruby tiene un truco elegante: los indices NEGATIVOS cuentan desde el final. El -1 es el ultimo elemento, el -2 el penultimo, y asi.

```
frutas = ["manzana", "platano", "naranja", "uva"]
#           0           1           2           3
#          -4          -3          -2          -1

puts frutas[-1] # "uva", el ultimo
puts frutas[-2] # "naranja", el penultimo
```

Esto es utilisimo cuando quieres el ultimo elemento pero no sabes cuantos hay. Sin indices negativos tendrias que hacer `frutas[frutas.length - 1]`, que es torpe y facil de equivocar. Con el -1, directo al grano.

Ruby también tiene metodos con nombre para los extremos, que a veces se leen mejor:

```
frutas.first # "manzana", igual que frutas[0]
frutas.last  # "uva", igual que frutas[-1]
```

Usa `.first` y `.last` cuando quieras dejar clarisima tu intencion ("quiero el primero/ultimo"); usa indices cuando la posicion sea variable o calculada.

Puedes incluso pedir varios elementos seguidos de formas que veremos en la seccion de rangos en indices. Por ahora, practica los indices simples hasta que el 0 inicial te salga natural. Un ejercicio mental: mira un arreglo y di en voz alta "el elemento 0 es X, el elemento -1 es Y". Repitelo con varios arreglos hasta que deje de sentirse raro.

Error comun: el desfase de uno al convertir "el tercero" a indice.

```
# "Quiero el tercer elemento"
frutas = ["manzana", "platano", "naranja"]
puts frutas[3] # MAL: nil, porque el tercero esta en el indice 2
puts frutas[2] # BIEN: "naranja"
```

La traduccion es: posicion humana N = indice N - 1. El primero es [0], el segundo es [1], el tercero es [2]. Pegate esta tabla en la mente o en un post-it hasta que sea automatica.

Rangos en índices: rebanar el arreglo

Puedes pedir no solo un elemento sino una REBANADA del arreglo usando un rango como índice. Es como cortar una rebanada de pastel: indicas desde donde hasta donde.

```
numeros = [10, 20, 30, 40, 50]

puts numeros[1..3].inspect # [20, 30, 40], del índice 1 al 3 (incluye el 3)
puts numeros[1...3].inspect # [20, 30], del 1 al 3 (EXCLUYE el 3)
puts numeros[0..2].inspect # [10, 20, 30]
```

La diferencia entre `..` (dos puntos) y `...` (tres puntos) es si el límite final se incluye o no. Con dos puntos, incluye el final; con tres puntos, lo excluye. Es un detalle pequeño con consecuencias grandes: ``1..3`` son tres elementos (1, 2, 3) y ``1...3`` son dos (1, 2). Cuando dudes, prueba con un ejemplo pequeño e imprime con `.inspect` para ver exactamente que obtuviste.

También puedes usar la forma índice, cantidad: ``arreglo[inicio, cantidad]``:

```
numeros = [10, 20, 30, 40, 50]
puts numeros[1, 3].inspect # [20, 30, 40], desde el 1, toma 3 elementos
```

Esta forma dice "empieza en el índice 1 y dame 3 elementos". Algunos la encuentran más clara que los rangos porque no hay que pensar en si el final se incluye. Ambas son correctas; usa la que te resulte más legible.

El método `.inspect` convierte el arreglo a su representación en texto, útil para depurar. Sin `inspect`, `puts` imprime cada elemento en una línea separada, lo que para verificar rebanadas es confuso. Acostúmbrate a usar `inspect` (o su atajo `p`) cuando quieras VER la estructura de un arreglo:

```
p numeros[1..3] # p es puts + inspect en uno solo
```

Error común: rebanadas que devuelven `nil` o arreglos vacíos inesperados.

```
numeros = [10, 20, 30]
p numeros[5..7] # nil, empieza fuera del arreglo
p numeros[2..10] # [30], se corta donde termina el arreglo, no truena
```


Ruby es tolerante: si la rebanada empieza fuera del arreglo, devuelve nil; si se pasa del final, devuelve hasta donde haya. No truena, pero puede darte resultados sorprendentes. Cuando una rebanada te devuelva algo raro, imprime el arreglo completo y los índices para ver donde quedo el corte.

Modificar arreglos: cambiar, agregar y quitar

Los arreglos en Ruby son mutables: puedes cambiar sus elementos, agregar nuevos y quitar los que no quieras. Para cambiar un elemento existente, lo asignas por índice:

```
frutas = ["manzana", "platano", "naranja"]
frutas[1] = "mango"
p frutas # ["manzana", "mango", "naranja"]
```

El platano fue reemplazado por el mango. Simple y directo.

Para AGREGAR al final hay dos formas equivalentes: push o el operador << (que la mayoría pronuncia "shovel", pala, porque parece una pala que mete cosas):

```
frutas = ["manzana"]
frutas.push("platano")
frutas << "naranja"
p frutas # ["manzana", "platano", "naranja"]
```

El << es el favorito de la comunidad por corto, y además se puede encadenar: `frutas << "uva" << "pera"`. El push también acepta varios a la vez: `frutas.push("uva", "pera")`. Ambos hacen lo mismo; elige uno y se consistente. Para QUITAR del final esta pop, que además te DEVUELVE el elemento quitado:

```
frutas = ["manzana", "platano", "naranja"]
ultima = frutas.pop
p frutas # ["manzana", "platano"]
puts ultima # "naranja"
```

El pop es como destapar una pila de platos: quita el de hasta arriba y te lo entrega en la mano. Es perfecto para estructuras tipo "el ultimo en entrar es el primero en salir".

Para el INICIO del arreglo estan unshift (agregar al inicio) y shift (quitar del inicio, devolviendo el elemento):

```

fila = ["Ana", "Luis"]
fila.unshift("Maria") # Maria llega y se pone al frente
p fila # ["Maria", "Ana", "Luis"]

primera = fila.shift # atienden a Maria, sale de la fila
p fila # ["Ana", "Luis"]
puts primera # "Maria"

```

Analogía completa: el arreglo es una fila de personas. push mete gente al final de la fila, pop saca a la última (la que acaba de llegar), unshift mete a alguien hasta adelante (como cuando dejan pasar a una embarazada), y shift atiende a la primera de la fila. Con estas cuatro operaciones puedes simular filas (shift para atender, push para formar) y pilas (push para apilar, pop para desapilar), que son estructuras de datos fundamentales.

Detalle de rendimiento que vale conocer: agregar/quitar al FINAL (push/pop) es rapidísimo; agregar/quitar al INICIO (unshift/shift) es más lento en arreglos grandes porque todos los demás elementos tienen que recorrerse una posición. Para listas pequeñas no importa, pero si algún día manejas miles de elementos y el programa se siente lento, este es un sospechoso habitual.

Error común: confundir el valor devuelto con el arreglo modificado.

```

frutas = ["manzana", "platano"]
resultado = frutas.pop
# resultado es "platano" (el elemento), NO el arreglo
# frutas ahora es ["manzana"]

```

El pop y el shift DEVUELVEN el elemento quitado y MODIFICAN el arreglo original. Si guardas el resultado esperando el arreglo, te llevas un string en lugar de un array y todo lo que sigue falla de formas confusas. Recuerda: estos métodos hacen dos cosas a la vez (modifican y devuelven), y tienes que tener claras las dos.

Otro error: usar delete cuando querías pop. El método delete quita TODAS las apariciones de un VALOR, no una posición:

```

numeros = [1, 2, 3, 2, 1]
numeros.delete(2)
p numeros # [1, 3, 1], quito TODOS los 2

```

Si solo querías quitar un 2, delete es demasiado agresivo. Para quitar por posición existe delete_at:

```
numeros = [1, 2, 3, 2, 1]
numeros.delete_at(1) # quita el elemento en el indice 1
p numeros # [1, 3, 2, 1]
```

Buena practica: cuando modifiques un arreglo dentro de un bucle que lo recorre, ten muchisimo cuidado, porque los indices se mueven bajo tus pies. Es una fuente legendaria de bugs. Si necesitas filtrar un arreglo, mejor construye uno nuevo con select (lo veremos) en lugar de borrar elementos mientras lo recorres con each.

map/collect: transformar cada elemento

Ya vimos que each sirve para HACER algo con cada elemento pero no para transformarlos. Para eso existe map: aplica el bloque a cada elemento y devuelve un ARREGLO NUEVO con los resultados. Su alias es collect; hacen exactamente lo mismo, pero map es el nombre que usa todo el mundo.

```
numeros = [1, 2, 3, 4]
dobles = numeros.map do |n|
  n * 2
end

p doubles # [2, 4, 6, 8], el arreglo nuevo
p numeros # [1, 2, 3, 4], el original NO cambio
```

El map no toca el original: construye uno nuevo. Esto es importante porque evita efectos secundarios sorpresa: puedes transformar datos sin miedo a arruinar la lista original. Si SI quieres modificar el original, existe map! con signo de exclamacion (los metodos con ! son la convencion Ruby para "version peligrosa que modifica el original"):

```
numeros = [1, 2, 3, 4]
numeros.map! do |n|
  n * 2
end
p numeros # [2, 4, 6, 8], el original SI cambio
```

Analogia: each es como un profesor que pasa lista y saluda a cada alumno (hace algo con ellos pero el grupo sigue igual); map es como un fotografo que le toma una foto a cada alumno y te entrega un album nuevo con las fotos (el grupo original no cambio, pero tienes una coleccion nueva transformada).

El map brilla con transformaciones de texto:

```

nombres = ["ana", "luis", "maria"]
mayusculas = nombres.map do |nombre|
  nombre.capitalize
end
p mayusculas # ["Ana", "Luis", "Maria"]

precios = [100, 250, 400]
con_iva = precios.map do |p|
  p * 1.16
end
p con_iva # [116.0, 290.0, 464.0]

```

También puedes encadenar metodos, que es donde Ruby se vuelve poesia:

```

resultado = [1, 2, 3, 4, 5]
  .select { |n| n % 2 == 0 }
  .map { |n| n * 10 }

p resultado # [20, 40]

```

Primero filtra los pares, luego multiplica por 10. Cada metodo devuelve un arreglo nuevo que el siguiente metodo procesa. Esta forma de encadenar es el estilo Ruby moderno y la veras en todo el codigo profesional.

Error comun numero uno: usar each cuando necesitas map y luego preguntarse por que el resultado es el arreglo original. Ya lo vimos en la seccion de each, pero vale repetirlo porque es EL error mas comun con estos dos metodos:

```

# MAL: each ignora lo que devuelve el bloque
nombres = ["ana", "luis"]
resultado = nombres.each { |n| n.capitalize }
p resultado # ["ana", "luis"], sin cambios!

# BIEN: map devuelve el arreglo transformado
resultado = nombres.map { |n| n.capitalize }
p resultado # ["Ana", "Luis"]

```

Error comun numero dos: olvidar que map devuelve un arreglo nuevo y no guardarlo en ningun lado:

```
# El resultado se calcula y se tira a la basura
numeros.map { |n| n * 2 }
p numeros # [1, 2, 3, 4], sin cambios, el map se desperdicio
```

Si no guardas el resultado del map en una variable (o lo usas directamente), el trabajo se pierde. El map sin asignacion es como cocinar una cena y tirarla: el esfuerzo se hizo pero nadie la aprovecho.

select/reject: filtrar elementos

El select recorre el arreglo y devuelve un ARREGLO NUEVO solo con los elementos donde el bloque devuelva verdadero. Es el filtro de la caja de herramientas. Su opuesto es reject, que devuelve los elementos donde el bloque devuelva FALSO (o sea, rechaza los que cumplan la condicion).

```
numeros = [1, 2, 3, 4, 5, 6]

pares = numeros.select do |n|
  n % 2 == 0
end
p pares # [2, 4, 6]

impares = numeros.reject do |n|
  n % 2 == 0
end
p impares # [1, 3, 5]
```

Fijate que select y reject con la MISMA condicion dan resultados complementarios: juntos cubren todo el arreglo original. Usa select cuando pienses "quiero los que cumplan X"; usa reject cuando pienses "quiero todos menos los que cumplan X". La diferencia es psicologica pero importa para la legibilidad: `select { |n| n % 2 == 0 }` vs `reject { |n| n % 2 != 0 }` dan lo mismo, pero el primero se lee directo y el segundo obliga a una doble negacion mental.

Ejemplo con datos reales: filtrar alumnos aprobados.

```
calificaciones = [95, 62, 78, 55, 88, 41]

aprobados = calificaciones.select { |c| c >= 70 }
reprobados = calificaciones.reject { |c| c >= 70 }

p aprobados    # [95, 78, 88]
p reprobados   # [62, 55, 41]
```

Como map, select y reject devuelven arreglos nuevos sin modificar el original (y tienen versiones con !: select! y reject!, que si modifican). Y como map, se encadenan de maravilla:

```
# Nombres de productos caros (mas de $500), en mayusculas
productos = ["laptop", "mouse", "monitor", "teclado"]
precios = [12000, 300, 4500, 800]

caros = precios.each_with_index
  .select { |precio, i| precio > 500 }
  .map { |precio, i| productos[i].upcase }

p caros # ["LAPTOP", "MONITOR", "TECLADO"]
```

(Este ejemplo usa each_with_index, que veremos en la siguiente seccion; por ahora solo aprecia el encadenamiento.)

Error comun: pensar que select modifica el original.

```
numeros = [1, 2, 3, 4, 5, 6]
numeros.select { |n| n % 2 == 0 }
p numeros # [1, 2, 3, 4, 5, 6], NO cambio!
```

Igual que con map: si no guardas el resultado, el filtro se desperdicia. O guardalo en una variable nueva, o usa select! si de verdad quieres modificar el original. La regla general de Ruby: los metodos sin ! devuelven algo nuevo y dejan el original intacto; los metodos con ! modifican el original. Hay excepciones (pop y shift modifican sin tener !), pero como guia funciona el 90% de las veces.

each_with_index: el elemento y su posicion

A veces necesitas el elemento Y su posicion al mismo tiempo: numerar una lista, mostrar "1. Manzana, 2. Platano", o procesar elementos según su lugar. Para eso existe each_with_index, que pasa DOS variables al bloque: el elemento y su indice.

```
frutas = ["manzana", "platano", "naranja"]

frutas.each_with_index do |fruta, i|
  puts "#{i + 1}. #{fruta}"
end
```

Imprime:

```
1. manzana
2. platano
3. naranja
```

Sumamos 1 al indice porque los humanos contamos desde 1 aunque Ruby cuente desde 0. Este patron de "indice + 1 para mostrar" lo usaras en cada lista numerada que presentes al usuario.

También sirve para trabajar con dos arreglos paralelos (dos listas donde la posicion conecta los datos):

```
nombres = ["Ana", "Luis", "Maria"]
calificaciones = [95, 82, 78]

nombres.each_with_index do |nombre, i|
  puts "#{nombre} saco #{calificaciones[i]}"
end
```

Aqui el indice *i* es el puente entre los dos arreglos: el nombre en la posicion *i* corresponde a la calificacion en la posicion *i*. Este patron de "arreglos paralelos" es comun en programas sencillos, aunque en programas mas grandes conviene usar hashes o objetos (los veras en otras unidades).

Con la forma de llaves en una linea también funciona:

```
frutas.each_with_index { |fruta, i| puts "#{i}: #{fruta}" }
```

Error comun: invertir el orden de las variables del bloque.

```
# MAL: el elemento va primero, el indice segundo
frutas.each_with_index do |i, fruta|
  puts "#{i}: #{fruta}"
end
# Imprime "manzana: 0", "platano: 1"... todo al reves
```

El orden es FIJO: primero el elemento, luego el indice. Si los inviertes, el programa no truena (Ruby no sabe que los nombraste mal), solo imprime tonterias. Cuando la salida se vea "volteada", revisa el orden de las variables del bloque.

length, size, empty?: medir el arreglo

Tres metodos basicos para preguntar por el tamano:

```
frutas = ["manzana", "platano", "naranja"]

puts frutas.length # 3
puts frutas.size   # 3, lo mismo que length
puts frutas.empty? # false

vacio = []
puts vacio.empty?  # true
```

length y size hacen exactamente lo mismo; son sinonimos. La mayoría usa length para arreglos y size para... bueno, también para arreglos. Usa el que prefieras, pero se consistente. El metodo empty? (con signo de interrogacion, la convencion Ruby para metodos que devuelven true/false) dice si el arreglo no tiene elementos.

Estos metodos son los guardianes de tus bucles y validaciones:

```
nombres = []

if nombres.empty?
  puts "La lista esta vacia, agrega nombres primero"
else
  puts "Hay #{nombres.length} nombres en la lista"
end
```

Y son la forma segura de recorrer con indices cuando de verdad necesitas indices (aunque recuerda: each es preferible):


```
i = 0
while i < frutas.length
  puts frutas[i]
  i = i + 1
end
```

La condicion `i < frutas.length` (con < estricto, no <=) es la forma correcta porque el ultimo indice valido es length - 1. Si usaras <=, intentarias leer frutas[3] en un arreglo de 3 elementos y obtendrias nil. Este es el famoso error "off by one" aplicado a arreglos: cuando un recorrido con indices falla en el ultimo elemento, revisa si usaste < o <=.

También existe count, que cuenta cuantos elementos cumplen una condicion:

```
numeros = [1, 2, 3, 4, 5, 6]
puts numeros.count { |n| n % 2 == 0 } # 3, hay tres pares
puts numeros.count # 6, sin bloque cuenta todos (igual que length)
```

El count con bloque es un select seguido de length, pero en un solo paso y mas legible: "cuenta cuantos son pares".

Arreglos anidados: tablas dentro de tablas

Un arreglo puede contener cualquier cosa, incluidos OTROS arreglos. A eso se le llama arreglo anidado, y es como Ruby representa tablas: cada arreglo interno es una fila.

```
alumnos = [
  ["Ana", 95, "A"],
  ["Luis", 82, "B"],
  ["Maria", 78, "C"]
]
```

Para acceder a un elemento interno usas DOS pares de corchetes: el primero elige la fila, el segundo la columna.

```
puts alumnos[0][0] # "Ana": fila 0, columna 0
puts alumnos[1][1] # 82: fila 1, columna 1
puts alumnos[2][2] # "C": fila 2, columna 2
```

Analogia: es como un edificio donde cada departamento tiene cuartos. El primer indice elige el departamento (la fila), el segundo elige el cuarto dentro de ese departamento (la columna). ``alumnos[1]`` te da la fila completa ``["Luis", 82, "B"]``, y ``alumnos[1][0]`` entra a esa fila y saca el primer dato.

Recorrer tablas anidadas con each es natural:

```
alumnos.each do |fila|
  nombre = fila[0]
  calificacion = fila[1]
  letra = fila[2]
  puts "#{nombre}: #{calificacion} (#{letra})"
end
```

O desempacando directamente en las variables del bloque (Ruby lo permite y se ve elegante):

```
alumnos.each do |nombre, calificacion, letra|
  puts "#{nombre}: #{calificacion} (#{letra})"
end
```

Cuando el bloque recibe un arreglo, Ruby reparte sus elementos entre las variables automaticamente. Es un truco bonito que hace el codigo mas legible: en lugar de `fila[0]`, `fila[1]`, tienes nombres descriptivos desde el inicio.

Puedes anidar mas niveles (arreglos de arreglos de arreglos), pero con dos niveles basta para el 95% de los casos. Si te encuentras con tres o mas niveles, probablemente tus datos piden una estructura mejor (hashes u objetos), que veras mas adelante en el curso.

Error comun: olvidar un nivel de corchetes.

```
puts alumnos[0]      # ["Ana", 95, "A"], la fila completa, no "Ana"
puts alumnos[0][0]   # "Ana", ahora si
```

Si esperabas un dato y te salio un arreglo completo entre corchetes, te faltó un nivel de indice. Y al reves: si pones un corchete de mas sobre algo que no es arreglo, truena con error. Cuenta tus niveles con calma.

split y chars: de texto a arreglo y de vuelta

Los strings y los arreglos son primos hermanos en Ruby, y hay dos metodos que los convierten entre si. El metodo split ROMPE un texto en pedazos y devuelve un arreglo; por defecto corta en los espacios:

```
frase = "hola mundo desde ruby"
palabras = frase.split
p palabras # ["hola", "mundo", "desde", "ruby"]
```

Puedes decirle por donde cortar pasandole un separador:

```
datos = "Ana,25,Mexico"
campos = datos.split(",")
p campos # ["Ana", "25", "Mexico"]

fecha = "2026-09-19"
partes = fecha.split("-")
p partes # ["2026", "09", "19"]
```

El split es la herramienta estandar para procesar datos en formato CSV (valores separados por comas), fechas, y cualquier texto con estructura. El separador se elimina del resultado: las comas no aparecen en el arreglo final.

El metodo chars rompe el texto en LETRAS individuales:

```
palabra = "hola"
letras = palabra.chars
p letras # ["h", "o", "l", "a"]
```

Esto sirve para analizar texto letra por letra: contar vocales, verificar palindromos, cifrar mensajes.

Y el camino de vuelta: join UNE un arreglo en un texto, con el separador que le digas:

```
palabras = ["hola", "mundo", "desde", "ruby"]
frase = palabras.join(" ")
puts frase # "hola mundo desde ruby"

letras = ["h", "o", "l", "a"]
palabra = letras.join
puts palabra # "hola", sin separador las pega directo
```

El trio split/join/chars forma un ciclo completo: texto -> arreglo -> texto. Lo usaras para procesar archivos, limpiar datos del usuario y transformar texto de mil formas. Ejemplo practico: invertir las palabras de una frase.

```
frase = "ruby es genial"
invertida = frase.split.reverse.join(" ")
puts invertida # "genial es ruby"
```

Lee la cadena de derecha a izquierda: split rompe en palabras, reverse las voltea, join las une con espacios. Tres metodos, una linea, cero bucles manuales. ESTO es programar en Ruby: componer metodos pequeños como piezas de Lego en lugar de escribir maquinaria desde cero.

Otro ejemplo: contar cuantas vocales tiene un texto.

```
texto = "programacion en ruby"
vocales = texto.chars.select do |letra|
  "aeiou".include?(letra)
end
puts "Vocales: #{vocales.length}" # 7
p vocales # ["o", "a", "a", "i", "o", "e", "u"]
```

chars convierte el texto en letras, select filtra las que estan en "aeiou", y length las cuenta. Fijate como se combinan las dos mitades de esta unidad: conversion de texto (chars) + filtrado (select) + medicion (length). Cada metodo hace una cosa simple; juntos resuelven problemas reales.

Error comun: olvidar que split sin argumentos corta por espacios multiples de forma inteligente, pero split(" ") no.

```
frase = "hola mundo"
p frase.split # ["hola", "mundo"], ignora espacios extra
p frase.split(" ") # ["hola", "", "", "mundo"], guarda los vacios
```

El split sin argumentos es mas listo: trata cualquier cantidad de espacios como un solo separador y quita los espacios de los extremos. Salvo que necesites el comportamiento literal, usa split sin argumentos para texto con espacios.

Ejemplos paso a paso

Ocho programas completos para practicar iteradores y arreglos. Como en la unidad anterior: copialos, corelos, rompelos a proposito y arreglalos. Cada error que provoques y entiendas es una leccion que no se te olvida.

Ejemplo 1: lista de compras interactiva. El programa deja agregar productos con push hasta que el usuario escriba "fin", y luego muestra la lista numerada con each_with_index. Junta entrada de usuario, bucle infinito controlado y arreglos.

```
compras = []

puts "Escribe tus productos (escribe 'fin' para terminar):"

while true
  print "> "
  producto = gets.chomp

  break if producto.downcase == "fin"

  if producto.empty?
    puts "Escribe algo, no dejes la linea vacia"
    next
  end

  compras << producto
end

puts ""
puts "Tu lista de compras (#{compras.length} productos):"

compras.each_with_index do |producto, i|
  puts "#{i + 1}. #{producto.capitalize}"
end
```

Analiza las piezas: el while true con break sale cuando el usuario escribe "fin" (comparado en minusculas con downcase para aceptar "FIN" o "Fin"). El next salta las lineas vacias sin agregarlas. El << agrega cada producto al final. Al final, each_with_index numera la lista sumando 1 al indice para mostrar. La variable compras vive FUERA del bucle para que no se reinicie en cada vuelta. Como extension, agrega la opcion de borrar un producto por numero usando delete_at.

Ejemplo 2: analizador de calificaciones. Pide calificaciones hasta que el usuario escriba -1, y luego calcula promedio, maxima, minima, aprobados y reprobados usando metodos de arreglos. Es el ejemplo perfecto de "juntar datos y luego analizarlos".

```
calificaciones = []

puts "Escribe calificaciones (0-100), -1 para terminar:"

while true
  print "> "
  cal = gets.chomp.to_f
  break if cal == -1

  if cal < 0 || cal > 100
    puts "Calificacion invalida, debe ser 0-100"
    next
  end

  calificaciones << cal
end

if calificaciones.empty?
  puts "No capturaste ninguna calificacion"
else
  suma = 0
  calificaciones.each { |c| suma = suma + c }
  promedio = suma / calificaciones.length

  puts "Capturaste #{calificaciones.length} calificaciones"
  puts "Promedio: #{promedio.round(2)}"
  puts "Maxima: #{calificaciones.max}"
  puts "Minima: #{calificaciones.min}"

  aprobados = calificaciones.select { |c| c >= 70 }
  puts "Aprobados: #{aprobados.length} (#{aprobados.join(", ")})"
end
```

Los metodos max y min devuelven el mayor y el menor del arreglo, directos y sin bucles. El promedio se calcula sumando con each y dividiendo entre length (también existe calificaciones.sum en Ruby moderno, que hace la suma en un paso). El select filtra los aprobados y join los muestra separados por comas. Fijate en la guardia del inicio: si el arreglo esta vacio, no intentamos dividir entre cero. Validar antes de calcular es un habito profesional.

Ejemplo 3: tabla de multiplicar con upto y map. Genera la tabla de un numero usando upto para los multiplicadores y map para construir las lineas de texto. Muestra como los iteradores reemplazan al while anidado de la unidad 3.

```
print "Que tabla quieres ver? "  
n = gets.chomp.to_i  
  
lineas = 1.upto(10).map do |i|  
  "#{n} x #{i} = #{n * i}"  
end  
  
puts "Tabla del #{n}:"  
puts lineas
```

El 1.upto(10) genera los numeros del 1 al 10, y el map convierte cada uno en un texto con el formato de la multiplicacion. El resultado es un arreglo de 10 strings que puts imprime uno por linea. Compara con la version de while anidado de la unidad anterior: esta version no tiene contadores manuales ni riesgo de olvidar el incremento. Cuando el numero de vueltas se conoce (10), el iterador es superior. Como reto, modifica el programa para que genere las tablas del 1 al 5 usando un upto externo y guardando todo en un arreglo de arreglos.

Ejemplo 4: buscador en lista con each y break. El programa tiene una lista de invitados y pregunta un nombre; lo busca con each y sale con break en cuanto lo encuentra. Usa una bandera como en la unidad 3.

```

invitados = ["Ana", "Luis", "Maria", "Pedro", "Sofia"]

print "Tu nombre: "
nombre = gets.chomp.capitalize

encontrado = false

invitados.each do |invitado|
  if invitado == nombre
    encontrado = true
    break
  end
end

if encontrado
  puts "Bienvenido, #{nombre}! Pasa a la fiesta"
else
  puts "Lo siento, #{nombre} no esta en la lista"
end

```

El break evita seguir buscando después de encontrar el nombre: para que recorrer mil invitados si el tuyo era el tercero. La bandera encontrado recuerda el resultado fuera del bucle. En Ruby real, este programa se escribiría en una línea con el método include?: `if invitados.include?(nombre)`. Pero entender la versión manual te enseña cómo funciona la búsqueda por dentro, y el patrón each + break + bandera aparece en problemas donde include? no basta (por ejemplo, buscar con condiciones complejas).

Ejemplo 5: cifrador Cesar con chars y map. El cifrado Cesar mueve cada letra N posiciones en el abecedario. Este programa lo implementa convirtiendo el texto en letras con chars, transformando cada una con map, y uniéndolo con join.


```

abecedario = ("a".."z").to_a

print "Texto a cifrar (minúsculas, sin acentos): "
texto = gets.chomp
print "Cuántas posiciones mover (1-25)? "
pasos = gets.chomp.to_i

cifrado = texto.chars.map do |letra|
  if abecedario.include?(letra)
    indice = abecedario.index(letra)
    nuevo_indice = (indice + pasos) % 26
    abecedario[nuevo_indice]
  else
    letra # espacios y signos se quedan igual
  end
end

puts "Cifrado: #{cifrado.join}"

```

Las piezas: `("a".."z").to_a` convierte el rango de letras en un arreglo real. `chars` rompe el texto en letras. `index` encuentra la posición de cada letra en el abecedario. El `% 26` hace que el desplazamiento "de la vuelta": la `z` con 3 pasos llega a la `c`, no se sale del arreglo. Las letras que no están en el abecedario (espacios, números) se dejan igual con el `else`. Prueba cifrar "hola" con 3 pasos: debe dar "krod". Y como reto, escribe el descifrador restando los pasos.

Ejemplo 6: carrito de compras con precios paralelos y `select`. Dos arreglos paralelos (productos y precios), un menú para agregar al carrito por número, y al final el total con `sum` y el producto más caro con `max`. Integra arreglos paralelos, `each_with_index` y métodos de agregación.

```

productos = ["Laptop", "Mouse", "Teclado", "Monitor", "Audifonos"]
precios    = [12000, 350, 800, 4500, 1200]

carrito = []

while true
  puts ""
  puts "--- TIENDA ---"
  productos.each_with_index do |prod, i|
    puts "#{i + 1}. #{prod} - ${precios[i]}"
  end
  puts "0. Terminar compra"
  print "Elige: "
  opcion = gets.chomp.to_i

  break if opcion == 0

  if opcion >= 1 && opcion <= productos.length
    carrito << (opcion - 1)
    puts "#{productos[opcion - 1]} agregado al carrito"
  else
    puts "Opcion invalida"
  end
end

if carrito.empty?
  puts "No compraste nada"
else
  puts ""
  puts "--- TU TICKET ---"
  total = 0

  carrito.each do |indice|
    puts "#{productos[indice]} - ${precios[indice]}"
    total = total + precios[indice]
  end

  puts "TOTAL: ${total}"

  mas_caro = carrito.map { |i| precios[i] }.max
  puts "Lo mas caro que compraste: ${mas_caro}"
end

```

El carrito guarda INDICES (posiciones) en lugar de nombres, y los precios se recuperan con esos indices: es el patron de arreglos paralelos en accion. El usuario elige con numeros del 1 al 5 (humanos, desde 1), y el programa resta 1 para obtener el indice real (opcion - 1): la traduccion humano-a-Ruby de la que hablamos. El map extrae solo los precios del carrito y max encuentra el mayor. Como mejora, cuenta cuantas veces se compro cada producto usando count.

Ejemplo 7: palindromo con chars, reverse y join. Un palindromo se lee igual al derecho y al revés ("anita lava la tina"). El programa limpia el texto (minusculas, sin espacios) y compara con su version invertida.

```
print "Escribe una frase: "
frase = gets.chomp.downcase

limpia = frase.chars.select do |letra|
  letra != " "
end
limpia = limpia.join

if limpia == limpia.reverse
  puts "'#{frase}' ES un palindromo"
else
  puts "'#{frase}' NO es un palindromo"
end
```

El metodo reverse invierte un string (o un arreglo). La logica es directa: si el texto limpio es igual a su reverso, es palindromo. El select quita los espacios para que "anita lava la tina" funcione. Prueba con "reconocer", "oso" y "hola". Como reto, haz que también ignore comas y puntos agregandolos a la condicion del select, o mejor: usa delete para quitar varios caracteres de un golpe.

Ejemplo 8: torneo con arreglo anidado y sort. Una tabla de equipos con puntos; el programa la ordena de mayor a menor con sort_by, muestra el podio con each_with_index y calcula estadísticas. Muestra arreglos anidados, ordenamiento y analisis.

```

equipos = [
  ["Tigres", 28],
  ["Aguilas", 35],
  ["Leones", 31],
  ["Tiburones", 22]
]

# Ordenar de mayor a menor puntaje
tabla = equipos.sort_by do |equipo|
  -equipo[1]
end

puts "--- TABLA FINAL ---"
tabla.each_with_index do |equipo, i|
  nombre = equipo[0]
  puntos = equipo[1]
  puts "#{i + 1}. #{nombre} - #{puntos} pts"
end

campeon = tabla.first
puts ""
puts "Campeon: #{campeon[0]} con #{campeon[1]} puntos"

total = equipos.map { |e| e[1] }.sum
puts "Puntos totales del torneo: #{total}"
puts "Promedio por equipo: #{(total.to_f / equipos.length).round(2)}"

```

El `sort_by` ordena según lo que devuelva el bloque; al negar los puntos (`-equipo[1]`) el orden queda descendente (truco clasico). `first` toma al campeon, `map` extrae los puntos y `sum` los totaliza. El `to_f` en la division evita que Ruby trunca los decimales (con enteros, $116 / 4$ da 29 pero $117 / 4$ da 29, no 29.25). Este ejemplo junta todo lo de la unidad: `anidados`, `each_with_index`, `map`, `first`, `sum`, `length`. Si entiendes este programa linea por linea, dominas la unidad.

Practica

Igual que antes: intenta cada ejercicio sin mirar la solucion. Si te atorras, mira, entiende, cierra y reescribe desde cero. Los ejercicios estan ordenados de menor a mayor dificultad.

Ejercicio 1: Usa `3.times` para imprimir "Ruby es genial" 3 veces, numerando cada linea (1, 2, 3). Recuerda que `times` empieza en 0.

Solucion explicada:

```
3.times do |i|  
  puts "#{i + 1}. Ruby es genial"  
end
```

La variable `|i|` toma los valores 0, 1, 2. Como los humanos numeramos desde 1, mostramos `i + 1`. Si imprimieras `i` directamente saldrían 0, 1, 2, que para un usuario final se ve raro. Este ajuste de `" + 1` para mostrar" aparece en cada lista numerada que presentes, así que conviértelo en reflejo.

Ejercicio 2: Pide un número `N` e imprime la cuenta regresiva desde `N` hasta 1 usando `downto`, y al final imprime "Feliz año nuevo!".

Solucion explicada:

```
print "Desde que numero contamos? "  
n = gets.chomp.to_i  
  
n.downto(1) do |i|  
  puts i  
end  
  
puts "Feliz año nuevo!"
```

El `downto` cuenta hacia abajo incluyendo ambos extremos: de `N` hasta 1. No hay contador manual que inicializar ni condición que pueda fallar: el iterador se encarga de todo. Compara con la versión `while`: necesitarías inicializar, comparar con `>= 1` y decrementar a mano, tres lugares donde equivocarte. Cuando el rango se conoce de antemano, el iterador gana siempre.

Ejercicio 3: Crea un arreglo con 5 de tus comidas favoritas. Imprime la primera y la última usando índices, y luego imprime todas numeradas con `each_with_index`.

Solucion explicada:

```
comidas = ["tacos", "pizza", "sushi", "enchiladas", "hamburguesa"]

puts "Primera: #{comidas[0]}"
puts "Ultima: #{comidas[-1]}"

comidas.each_with_index do |comida, i|
  puts "#{i + 1}. #{comida}"
end
```

El indice 0 da la primera y el -1 la ultima sin necesidad de saber cuantos elementos hay. El `each_with_index` numera mostrando `i + 1`. Tres conceptos de la unidad en un ejercicio pequeno: creacion de arreglos, indices positivos/negativos e iteracion con posicion. Si esto te sale natural, la base esta solida.

Ejercicio 4: Dado el arreglo [3, 7, 2, 9, 4, 6], crea un arreglo nuevo con solo los numeros mayores que 5 usando `select`, y otro con esos mismos numeros multiplicados por 10 usando `map`. Encadena ambos en una sola linea como reto extra.

Solucion explicada:

```
numeros = [3, 7, 2, 9, 4, 6]

mayores = numeros.select { |n| n > 5 }
p mayores # [7, 9, 6]

por_diez = mayores.map { |n| n * 10 }
p por_diez # [70, 90, 60]

# Reto extra: encadenado en una sola expresion
resultado = numeros.select { |n| n > 5 }.map { |n| n * 10 }
p resultado # [70, 90, 60]
```

El `select` filtra primero (devuelve [7, 9, 6] sin tocar el original) y el `map` transforma ese resultado. En la version encadenada, el arreglo que devuelve `select` entra directo al `map`: no hace falta la variable intermedia. Lee la cadena de izquierda a derecha como una tuberia: los numeros entran, se filtran los mayores que 5, se multiplican por 10, y sale el resultado. Esta forma de pensar en "tuberias de datos" es el corazon del estilo Ruby.

Ejercicio 5: Pide al usuario 5 nombres, guardalos en un arreglo con `<<`, y luego muestra: cuantos nombres hay, el primero en orden alfabetico (`sort`) y todos en mayusculas (`map`) sin modificar el original.

Solucion explicada:

```
nombres = []

5.times do
  print "Escribe un nombre: "
  nombres << gets.chomp
end

puts "Capturaste #{nombres.length} nombres"
puts "Primero alfabeticamente: #{nombres.sort.first}"

mayusculas = nombres.map { |n| n.upcase }
puts "En mayusculas: #{mayusculas.join(", ")}"
puts "Original intacto: #{nombres.join(", ")}"
```

El 5.times repite la pregunta 5 veces sin contador manual, y el << agrega cada nombre al final. sort ordena alfabeticamente (devuelve un arreglo nuevo ordenado) y first toma el primero. El map con upcase crea la version en mayusculas dejando el original intacto, y el ultimo puts lo demuestra. Fijate en el patron general: recolectar con << en un bucle, y analizar con metodos después. Es el flujo de trabajo estandar con arreglos.

Ejercicio 6: Convierta la frase "el ruby es divertido" en un arreglo de palabras con split, invierte el orden con reverse, vuelve a unir con join, y muestra el resultado. Todo en una sola linea.

Solucion explicada:

```
frase = "el ruby es divertido"
resultado = frase.split.reverse.join(" ")
puts resultado # "divertido es ruby el"
```

Tres metodos encadenados: split rompe en ["el", "ruby", "es", "divertido"], reverse lo voltea a ["divertido", "es", "ruby", "el"], y join(" ") lo une con espacios. Ningun bucle manual, ninguna variable temporal. Si puedes leer esta linea de izquierda a derecha y predecir el resultado, ya piensas en Ruby: componer transformaciones pequenas en lugar de escribir maquinaria.

Ejercicio 7: Crea un arreglo anidado con 3 alumnos (nombre y calificacion). Recorrelo con each desempacando las variables en el bloque, calcula el promedio, y muestra quien saco la mejor calificacion usando max_by.

Solucion explicada:

```

alumnos = [
  ["Ana", 95],
  ["Luis", 82],
  ["Maria", 91]
]

alumnos.each do |nombre, calificacion|
  puts "#{nombre}: #{calificacion}"
end

suma = alumnos.map { |nombre, calificacion| calificacion }.sum
promedio = suma.to_f / alumnos.length
puts "Promedio del grupo: #{promedio.round(2)}"

mejor = alumnos.max_by { |nombre, calificacion| calificacion }
puts "Mejor calificacion: #{mejor[0]} con #{mejor[1]}"

```

El bloque `|nombre, calificacion|` desempaca cada fila automáticamente: Ruby reparte los elementos del arreglo interno entre las variables. El `map` extrae solo las calificaciones y sum las totaliza; el `to_f` evita la division entera. El `max_by` devuelve el ELEMENTO completo (la fila `["Ana", 95]`) cuyo bloque dio el valor maximo, por eso accedemos con `mejor[0]` y `mejor[1]`. Es el primo de `sort_by` especializado en encontrar el maximo sin ordenar todo.

Ejercicio 8: Simula una fila del banco: usa un arreglo como cola con `push` para formar personas y `shift` para atenderlas. Forma a 4 personas, atiende a 2 mostrando a quien atiendes, y muestra quienes quedan en la fila. Luego explica por que `push/shift` modela una fila y `push/pop` modelaria una pila.

Solucion explicada:


```

fila = []

fila.push("Ana")
fila.push("Luis")
fila.push("Maria")
fila.push("Pedro")
puts "Fila inicial: #{fila.join(", ")}"

2.times do
  atendido = fila.shift
  puts "Atendiendo a: #{atendido}"
end

puts "Quedan en la fila: #{fila.join(", ")}"

```

El push agrega al final (la gente se forma atras) y el shift saca del inicio (se atiende al primero): eso es una cola o fila, "el primero en llegar es el primero en salir". Si en cambio usaras pop, sacarías al ULTIMO en llegar, que es una pila ("el ultimo en entrar es el primero en salir", como una pila de platos). La diferencia entre fila y pila es solo que extremo usas para sacar: shift = fila, pop = pila. Este modelo aparece en sistemas de turnos, impresoras, y atencion a clientes: la teoria de colas de la vida real, en 10 lineas de Ruby.

Resumen de la unidad

- times repite N veces (empieza en 0), upto cuenta hacia arriba y downto hacia abajo. Cuando el numero de vueltas se conoce de antemano, usa iteradores en lugar de while: menos tramite, cero riesgo de bucle infinito.
- each es el iterador estrella: recorre cada elemento sin manejar indices. Sirve para HACER algo con cada elemento, no para transformar (para eso esta map).
- Prefiere each sobre for: es lo idiomatico en Ruby y sus variables de bloque no se escapan al resto del programa.
- Los arreglos guardan listas ordenadas entre corchetes. Los indices empiezan en 0; los negativos cuentan desde el final (-1 es el ultimo). first y last son atajos legibles para los extremos.
- Con rangos en los indices sacas rebanadas: arreglo[1..3] incluye el 3, arreglo[1...3] lo excluye. La forma arreglo[inicio, cantidad] es la alternativa sin ambigüedad.
- push y << agregan al final, pop quita del final devolviendo el elemento; unshift agrega al inicio, shift quita del inicio devolviendolo. push/shift modela filas, push/pop modela pilas.

- map/collect devuelve un arreglo NUEVO transformando cada elemento; select filtra los que cumplan la condicion y reject los que no. Guarda el resultado o se desperdicia; las versiones con ! modifican el original.
- each_with_index te da el elemento y su posicion: ideal para numerar listas y conectar arreglos paralelos.
- length/size miden, empty? pregunta si esta vacio, count cuenta con condicion, max/min dan los extremos, sum totaliza, sort ordena, reverse invierte.
- Los arreglos anidados representan tablas: dos pares de corchetes para llegar al dato (fila y columna), y el bloque de each puede desempacar las variables directamente.
- split rompe texto en arreglos, chars lo rompe en letras, join une arreglos en texto. Encadenarlos (split.reverse.join) resuelve problemas de texto sin bucles manuales.
- gets entrega texto: acuerdate de to_i/to_f para numeros y downcase para comparar texto del usuario sin pelear con las mayusculas.

Bloques: la pieza que hace funcionar a los iteradores

Ya usaste bloques en toda esta unidad: todo lo que va entre do...end o entre llaves {} después de un metodo como each, map o times es un bloque. Pero vale la pena entender que SON, porque son la idea mas distintiva de Ruby y la razon por la que sus iteradores se sienten magicos.

Un bloque es un pedazo de codigo que le ENTREGAS a un metodo para que el metodo lo ejecute cuando quiera, las veces que quiera. Cuando escribes:

```
3.times do |i|
  puts "Vuelta #{i}"
end
```

Le estas diciendo al metodo times: "toma este pedazo de codigo y ejecutalo 3 veces, pasandole el numero de vuelta en la variable i". El times decide CUANTAS veces se ejecuta; tu decides QUE se ejecuta. Esa division del trabajo es poderosissima: los metodos manejan la mecanica (contar, recorrer, filtrar) y tu solo describes la accion.

Las variables entre barras verticales |i|, |fruta|, |n| se llaman parametros del bloque: son los valores que el metodo le pasa a tu codigo en cada ejecucion. El each te pasa cada elemento; el each_with_index te pasa el elemento y el indice; el times te pasa el numero de vuelta. Tu los nombras como quieras: |x| funciona igual que |fruta|, pero los nombres descriptivos hacen el codigo legible.

Las dos sintaxis (do...end y llaves) son intercambiables para el lenguaje, pero la comunidad tiene una convencion clara: llaves {} para bloques de una sola linea, do...end para bloques de varias lineas.

```
# Una linea: llaves
nombres.each { |n| puts n }

# Varias lineas: do...end
nombres.each do |n|
  puts "Hola, #{n}"
  puts "Bienvenido al curso"
end
```

Seguir esta convencion no es obligatorio, pero hace que tu codigo se vea como el de un programador Ruby experimentado. Y hay una diferencia sutil de prioridad entre ambas que causa bugs raros en codigo avanzado; por ahora basta con saber la convencion y aplicarla.

Un detalle hermoso: como los bloques son codigo que se pasa a metodos, puedes guardar logica reutilizable de formas muy expresivas. Por ejemplo, este patron aparece en todas las aplicaciones Ruby del mundo:

```
# "Haz esto con cada usuario admin"
usuarios.select { |u| u[:admin] }.each { |u| puts u[:nombre] }
```

Lee la linea como una frase: "de los usuarios, selecciona los admin, y con cada uno imprime su nombre". Cada bloque es una mini-instruccion en espanol casi literal. Cuando tu codigo se lee como una frase, esta bien escrito.

Error comun: olvidar las barras verticales o poner mal los parametros.

```
# MAL: sin las barras, i no existe dentro del bloque
3.times do
  puts "Vuelta #{i}"
end
```

Sin `|i|` en la declaracion del bloque, la `i` de adentro es una variable desconocida (nil o error, según el caso). Las barras no son decoracion: son las que conectan tu codigo con los valores que el metodo te pasa. Si un iterador "no te da" el valor esperado, revisa primero las barras y los nombres de los parametros.

Mas metodos utiles de arreglos: la caja sigue creciendo

Ya dominas lo esencial, pero los arreglos en Ruby tienen decenas de metodos mas. Estos son los que mas vas a usar en la vida real:

`include?` pregunta si un valor esta en el arreglo (devuelve true/false):

```
invitados = ["Ana", "Luis", "Maria"]
puts invitados.include?("Luis")  # true
puts invitados.include?("Pedro") # false
```

Es la forma idiomática de buscar: reemplaza al `each` + bandera + `break` que hicimos a mano en los ejemplos. Usa la versión manual para aprender como funciona por dentro; usa `include?` en tu código de todos los días.

`uniq` elimina duplicados devolviendo un arreglo nuevo:

```
numeros = [1, 2, 2, 3, 3, 3, 4]
p numeros.uniq # [1, 2, 3, 4]
```

`flatten` aplana arreglos anidados en uno solo:

```
anidado = [1, [2, 3], [4, [5, 6]]]
p anidado.flatten # [1, 2, 3, 4, 5, 6]
```

Útil cuando juntas datos de varias fuentes y te quedan listas dentro de listas sin querer.

`sort` ordena (números de menor a mayor, texto alfabéticamente):

```
p [3, 1, 2].sort          # [1, 2, 3]
p ["pera", "manzana"].sort # ["manzana", "pera"]
```

Y `sort_by` ordena según un criterio que tu defines con un bloque:

```
palabras = ["a", "elefante", "sol", "murcielago"]
p palabras.sort_by { |p| p.length } # ["a", "sol", "elefante", "murcielago"]
```

`sample` toma elementos al azar y `shuffle` los revuelve:

```
p [1, 2, 3, 4, 5].sample # un elemento aleatorio, ej: 3
p [1, 2, 3, 4, 5].shuffle # orden aleatorio, ej: [4, 1, 5, 2, 3]
```

Perfectos para juegos, sorteos y barajar cartas.

`delete_if` borra los elementos que cumplan la condicion del bloque (modifica el original):

```
numeros = [1, 2, 3, 4, 5, 6]
numeros.delete_if { |n| n % 2 == 0 }
p numeros # [1, 3, 5]
```

Es el primo destructivo de `reject`: en lugar de devolver un arreglo nuevo sin esos elementos, los quita del original. Recuerda la regla del signo de exclamacion... aunque este no lleva ! y aun asi modifica: es una de esas excepciones que confirman la regla. Cuando un metodo modifique el original sin avisar con !, la documentacion (o la experiencia) te lo enseñara; ante la duda, prueba con un ejemplo pequeno e imprime antes y después.

`zip` combina dos arreglos elemento por elemento en pares:

```
nombres = ["Ana", "Luis", "Maria"]
edades = [25, 30, 22]
p nombres.zip(edades) # [{"Ana", 25}, {"Luis", 30}, {"Maria", 22}]
```

Convierte dos arreglos paralelos en un arreglo anidado de pares: justo la estructura de tabla que vimos en la seccion de anidados. Es el puente entre "dos listas separadas" y "una tabla", y aparece mucho al combinar datos de distintas fuentes.

Cazando bugs de arreglos: guia de supervivencia

Los arreglos generan errores muy caracteristicos. Aprende a reconocerlos y los resolveras en minutos en lugar de horas.

Sintoma 1: "undefined method X for nil" (metodo no definido para nil). Traduccion: intentaste llamar un metodo sobre nil, lo que casi siempre significa que un indice te devolvio nil porque pediste una posicion inexistente.

```
frutas = ["manzana", "platano"]
puts frutas[5].upcase # ERROR: upcase no existe para nil
```

Cura: imprime el valor antes de usarlo (`p frutas[5]`) y verifica el indice. El 90% de las veces contaste desde 1 o el arreglo tiene menos elementos de los que crees.

Sintoma 2: el bucle "solo funciona la primera vez" o "se salta elementos". Casi siempre es una variable no reiniciada en bucles anidados, o peor: modificar el arreglo mientras lo recorres.

```
# MAL: borrar mientras recorres mueve los indices bajo tus pies
numeros = [1, 2, 3, 4, 5, 6]
numeros.each do |n|
  numeros.delete(n) if n % 2 == 0
end
p numeros # resultado impredecible, probablemente [1, 3, 5]... o no
```

Cura: nunca modifiques la coleccion que estas recorriendo. Construye una nueva con select/reject, o itera sobre una copia con numeros.dup.each. La regla es simple: el recorrido lee, otro metodo escribe.

Sintoma 3: el map/select "no hizo nada". El resultado se calculo pero nadie lo guardo.

```
nombres = ["ana", "luis"]
nombres.map { |n| n.capitalize }
p nombres # ["ana", "luis"], sin cambios
```

Cura: guarda el resultado (nombres = nombres.map {...}) o usa la version con ! si de verdad quieres modificar el original. Cada vez que un metodo "no funcione", pregunta: "donde quedo lo que devolvio?".

Sintoma 4: comparaciones que fallan misteriosamente con datos del usuario.

```
print "Escribe tu nombre: "
nombre = gets.chomp
invitados = ["Ana", "Luis"]

if invitados.include?(nombre)
  puts "Bienvenido"
else
  puts "No estas en la lista" # sale esto aunque escribas "Ana"
end
```

Si escribiste "ana" con minuscula, include? dice false porque "ana" != "Ana". Cura: normaliza antes de comparar (nombre.capitalize o comparar todo en minusculas). Los datos del usuario SIEMPRE llegan sucios: espacios extra, mayusculas raras. Limpialos con strip (quita espacios de los extremos), downcase o capitalize antes de procesarlos.

Sintoma 5: division entera que "pierde" decimales al calcular promedios.

```
calificaciones = [95, 82, 78]
promedio = calificaciones.sum / calificaciones.length
puts promedio # 85, pero el real es 85.0... aqui coincide, pero con [95, 82] daria
88 en vez de 88.5
```

Cura: convierte a flotante antes de dividir (`calificaciones.sum.to_f / calificaciones.length`). Cada vez que un promedio salga "redondo" sospechosamente, revisa si dividiste enteros entre enteros.

La tecnica universal de depuracion para arreglos es `p` (`inspect`): imprime el arreglo en los puntos clave y compara lo que ESPERAS con lo que HAY. La mayoría de los bugs de arreglos se revelan en cuanto VES el contenido real en lugar del que imaginabas.

Ejemplos paso a paso (segunda parte)

Dos programas mas grandes que integran todo lo de la unidad. Son el examen final practico: si los entiendes completos, la unidad esta dominada.

Ejemplo 9: analizador de texto completo. Pide un parrafo y reporta: numero de palabras, palabra mas larga, palabras unicas ordenadas, y cuantas veces aparece cada vocal. Es procesamiento de texto real con `split`, `map`, `uniq`, `sort` y `chars`.

```

print "Escribe un parrafo corto: "
texto = gets.chomp.downcase

palabras = texto.split
puts "Total de palabras: #{palabras.length}"

mas_larga = palabras.max_by { |p| p.length }
puts "Palabra mas larga: '#{mas_larga}' (#{mas_larga.length} letras)"

unicas = palabras.uniq.sort
puts "Palabras unicas (#{unicas.length}): #{unicas.join(", ")}"

vocales = "aeiou"
conteo = {}

vocales.chars.each do |v|
  cantidad = texto.chars.count { |letra| letra == v }
  conteo[v] = cantidad
end

puts "Vocales:"
conteo.each do |vocal, cantidad|
  puts "  #{vocal}: #{cantidad}"
end

```

Analisis por partes: split rompe el parrafo en palabras; max_by encuentra la mas larga por longitud; uniq quita repetidas y sort las ordena para el reporte. Para las vocales usamos un hash (los estudiaras a fondo en otra unidad, pero aqui lo ves en accion simple): por cada vocal contamos sus apariciones con chars.count y guardamos el par en el hash. El ultimo each recorre el hash mostrando los resultados. Este programa parece avanzado pero cada linea usa un metodo de esta unidad: es la prueba de que ya tienes las piezas para resolver problemas reales.

Ejemplo 10: juego de memoria con shuffle. Muestra una secuencia de palabras por unos segundos, la oculta, y el usuario debe repetirla en orden. Usa shuffle para aleatoriedad, sleep para la pausa, y comparacion de arreglos con ==.


```

palabras = %w(sol luna estrella nube rio monte)
ronda = palabras.sample(4)

puts "Memoriza estas palabras en orden:"
puts ronda.join(" - ")
puts "(Tienes 5 segundos...)"
sleep(5)

puts ""
puts "" * 30
puts "Se acabo el tiempo! Escribelas en orden, separadas por espacios:"
print "> "
respuesta = gets.chomp.downcase.split

if respuesta == ronda
  puts "Perfecto! Memoria de elefante"
else
  puts "Casi. El orden correcto era: #{ronda.join(" - ")}"

  aciertos = 0
  respuesta.each_with_index do |palabra, i|
    if ronda[i] == palabra
      aciertos = aciertos + 1
      puts " Posicion #{i + 1}: bien (#{palabra})"
    else
      puts " Posicion #{i + 1}: era '#{ronda[i]}', pusiste '#{palabra}'"
    end
  end
  puts "Acertaste #{aciertos} de #{ronda.length}"
end

```

Las piezas: %w crea el banco de palabras; sample(4) toma 4 al azar (con argumento, sample devuelve un arreglo); sleep(5) pausa 5 segundos; los puts vacios simulan limpiar la pantalla. La comparacion respuesta == ronda funciona porque en Ruby dos arreglos son iguales si tienen los mismos elementos en el mismo orden. El each_with_index da retroalimentacion posicion por posicion. Ojo: ronda[i] puede ser nil si el usuario escribio menos palabras; el programa no truena, solo muestra nil en el mensaje. Como mejora, valida que la respuesta tenga 4 palabras antes de calificar.

Practica (segunda parte)

Ejercicio 9: Dado el arreglo ["ana", "LUIS", "Maria", "pedro"], normaliza todos los nombres a formato capitalizado ("Ana", "Luis", ...) usando map, sin modificar el original. Luego verifica con include? si "Luis" esta en el arreglo normalizado.

Solucion explicada:

```
nombres = ["ana", "LUIS", "Maria", "pedro"]

normalizados = nombres.map { |n| n.capitalize }
p normalizados # ["Ana", "Luis", "Maria", "Pedro"]
p nombres     # ["ana", "LUIS", "Maria", "pedro"], intacto

puts normalizados.include?("Luis") # true
```

El capitalize convierte la primera letra a mayuscula y el resto a minusculas, justo lo que necesitas para normalizar. El map devuelve el arreglo nuevo y el original queda intacto, como demuestra el segundo p. El include? confirma que "Luis" ya esta en formato buscable. Este es el flujo estandar de limpieza de datos: normalizar primero, buscar después. Apicalo a todo dato que venga del usuario.

Ejercicio 10: Crea un programa que pida 6 numeros, los guarde en un arreglo, y luego muestre un mini-reporte: el arreglo ordenado, sin duplicados, el mayor, el menor, la suma, el promedio (con decimales) y cuantos son pares. Usa un metodo distinto para cada dato.

Solucion explicada:

```
numeros = []

6.times do
  print "Escribe un numero: "
  numeros << gets.chomp.to_i
end

puts "Ordenados: #{numeros.sort.inspect}"
puts "Sin duplicados: #{numeros.uniq.inspect}"
puts "Mayor: #{numeros.max}"
puts "Menor: #{numeros.min}"
puts "Suma: #{numeros.sum}"
puts "Promedio: #{(numeros.sum.to_f / numeros.length).round(2)}"
puts "Pares: #{numeros.count { |n| n % 2 == 0 }}"
```

Cada linea del reporte usa un metodo de la unidad: sort, uniq, max, min, sum, count con bloque. El promedio convierte la suma a flotante con `to_f` antes de dividir para no perder decimales, y `round(2)` lo deja en 2 decimales. El `6.times` recolecta sin contador manual. Este ejercicio es un catalogo viviente de metodos de arreglos: si lo escribes sin mirar, los tienes en la memoria muscular.

each vs map vs select: la guia definitiva de cuando usar cual

Estos tres metodos son los que mas se confunden entre si, asi que vamos a dejar las diferencias cristalinas con una regla mental para cada uno. Piensa en una fabrica de juguetes con una banda transportadora:

- `each` es el INSPECTOR: mira cada juguete, hace algo con el (lo sella, lo cuenta, lo anuncia por la bocina), pero la banda sigue igual al final. Lo usas por sus EFECTOS: imprimir, guardar, enviar, acumular en una variable externa. Devuelve el arreglo original porque su trabajo no era transformarlo.

```
# Inspector: anuncia cada juguete, la banda no cambia
juguetes.each { |j| puts "Revisando: #{j}" }
```

- `map` es el PINTOR: toma cada juguete y te entrega uno NUEVO pintado de otro color. La banda original queda intacta; lo que te importa es la banda nueva de juguetes transformados. Lo usas por su RESULTADO: un arreglo nuevo del mismo tamaño con cada elemento transformado.

```
# Pintor: devuelve juguetes nuevos, uno por cada original
pintados = juguetes.map { |j| "rojo #{j}" }
```

- `select` es el FILTRO: deja pasar solo los juguetes que cumplen la condicion y te entrega la banda filtrada. Lo usas por su RESULTADO: un arreglo nuevo (casi siempre mas chico) con solo los elementos elegidos.

```
# Filtro: solo pasan los que cumplen la condicion
elegidos = juguetes.select { |j| j.length > 5 }
```

La pregunta magica para elegir es: "que quiero al final?". Si quieres EFECTOS (que pase algo), `each`. Si quieres los MISMOS elementos TRANSFORMADOS, `map`. Si quieres MENOS elementos (los que cumplan algo), `select`. Y si quieres efectos Y filtrar, los encadenas: primero `select`, luego `each`.

```
# Efectos sobre un subconjunto: filtra primero, actua después
juguetes.select { |j| j.length > 5 }.each { |j| puts "Empacando: #{j}" }
```

Un error sutil: usar map solo por sus efectos e ignorar el resultado.

```
# Funciona pero es mala practica: map crea un arreglo que nadie usa
nombres.map { |n| puts n }
```

Esto imprime los nombres (porque puts esta en el bloque), pero también construye un arreglo de nils que se tira a la basura. Para solo imprimir, each es lo correcto: comunica tu intencion real. El codigo no solo debe funcionar: debe decir lo que quiere decir. Un programador que lea `map` esperara una transformacion; si solo querias imprimir, lo confundiste gratis.

Otro error sutil: acumular a mano lo que un metodo ya hace.

```
# Torpe: each con acumulador manual
dobles = []
numeros.each { |n| doubles << n * 2 }

# Directo: map hace exactamente eso
dobles = numeros.map { |n| n * 2 }
```

Cada vez que veas un each que solo llena un arreglo nuevo con <<, es un map disfrazado. Reconocer estos patrones es lo que convierte codigo de principiante en codigo Ruby idiomatico. No es esnobismo: el map dice en una linea lo que el each dice en tres, y menos lineas son menos lugares donde equivocarse.

Proyecto integrador: sistema de calificaciones completo

Para cerrar la unidad, un proyecto que usa casi todo lo aprendido en las unidades 3 y 4. Es mas largo que los ejemplos, asi que leelo por secciones y asegurate de entender cada una antes de pasar a la siguiente. Copialo, correlo, y luego modificalo: agrega la opcion de borrar un alumno, o de guardar el reporte en un archivo.

```
/sistema de calificaciones - proyecto integrador unidad 4/

alumnos = [] # arreglo anidado: [nombre, calificacion]

while true
  puts ""
  puts "=== SISTEMA DE CALIFICACIONES ==="
  puts "1. Agregar alumno"
```

```

puts "2. Ver todos"
puts "3. Buscar alumno"
puts "4. Reporte del grupo"
puts "5. Salir"
print "Opcion: "
opcion = gets.chomp

case opcion
when "1"
  print "Nombre: "
  nombre = gets.chomp.capitalize
  print "Calificacion (0-100): "
  cal = gets.chomp.to_f

  if cal < 0 || cal > 100
    puts "Calificacion invalida"
  else
    alumnos << [nombre, cal]
    puts "#{nombre} agregado con #{cal}"
  end
end

when "2"
  if alumnos.empty?
    puts "No hay alumnos registrados"
  else
    puts "--- LISTA DE ALUMNOS ---"
    alumnos.each_with_index do |(nombre, cal), i|
      estado = cal >= 70 ? "aprobado" : "reprobado"
      puts "#{i + 1}. #{nombre}: #{cal} (#{estado})"
    end
  end
end

when "3"
  print "Nombre a buscar: "
  buscado = gets.chomp.capitalize
  encontrado = alumnos.select { |(nombre, cal)| nombre == buscado }

  if encontrado.empty?
    puts "#{buscado} no esta registrado"
  else
    encontrado.each do |(nombre, cal)|
      puts "#{nombre}: #{cal}"
    end
  end
end

```

```

when "4"
  if alumnos.empty?
    puts "No hay datos para el reporte"
  else
    calificaciones = alumnos.map { |(nombre, cal)| cal }
    promedio = calificaciones.sum.to_f / calificaciones.length
    aprobados = calificaciones.count { |c| c >= 70 }

    puts "--- REPORTE DEL GRUPO ---"
    puts "Total de alumnos: #{alumnos.length}"
    puts "Promedio: #{promedio.round(2)}"
    puts "Calificacion mas alta: #{calificaciones.max}"
    puts "Calificacion mas baja: #{calificaciones.min}"
    puts "Aprobados: #{aprobados} de #{alumnos.length}"

    mejor = alumnos.max_by { |(nombre, cal)| cal }
    puts "Mejor alumno: #{mejor[0]} con #{mejor[1]}"
  end

when "5"
  puts "Adios!"
  break

else
  puts "Opcion invalida"
end
end

```

Que usa de la unidad 3: while true con break para el menu, case/when para las opciones, if/else para validaciones, el ternario para el estado aprobado/reprobado, y la guardia empty? antes de procesar. Que usa de la unidad 4: arreglo anidado para los alumnos, push (<<) para agregar, each_with_index con desempacado para listar, select para buscar, map para extraer calificaciones, sum/max/min/count para estadisticas, max_by para el mejor alumno.

Fijate en el desempacado en el bloque: |(nombre, cal)|, |i| combina el desempacado de la fila con el indice de each_with_index. Es sintaxis avanzada pero legible una vez que conoces las piezas. Y nota como cada opcion del menu valida primero (empty?, rangos) antes de procesar: programar a la defensiva, esperando lo peor del usuario, es lo que separa un programa de juguete de uno que aguanta el mundo real.

Como retos para seguir practicando: agrega la opcion 6 para borrar un alumno por nombre (usa `delete_if` o `reject!`), la opcion 7 para mostrar solo los reprobados ordenados de menor a mayor (`select + sort_by`), y haz que el reporte muestre también la mediana además del promedio. Cada reto usa metodos de esta unidad: si los resuelves sin mirar atras, estas listo para la siguiente unidad.

Con esto cierras la unidad 4. Ya sabes tomar decisiones, repetir acciones de todas las formas idiomáticas de Ruby, y manejar colecciones de datos como un profesional: crearlas, recorrerlas, transformarlas, filtrarlas y analizarlas. En la siguiente unidad entrarás a los hashes y a los metodos, donde todo esto se combina en programas mas grandes y organizados. Sigue practicando: la diferencia entre leer sobre map y dominar map son unos veinte programas escritos con tus propias manos.

Lo que sigue

Con las unidades 3 y 4 ya tienes el nucleo de la programacion imperativa: decidir con condicionales, repetir con bucles e iteradores, y manejar colecciones con arreglos. Todo programa que escribas de aqui en adelante, sin importar que tan grande sea, estara construido con estas piezas. Los frameworks, las aplicaciones web y los videojuegos no usan magia distinta: usan `if`, `each`, `map` y arreglos, combinados a gran escala.

Antes de pasar a la siguiente unidad, hazte este examen honesto de tres preguntas. Primera: puedes escribir un menu interactivo con `while true`, `case/when` y `break` sin mirar tus apuntes? Segunda: dada una lista de numeros, puedes en una sola linea encadenada quedarte con los pares, duplicarlos y sumarlos? Tercera: puedes explicar con tus palabras por que en Ruby solo `false` y `nil` son falsos, y que consecuencias tiene eso al validar datos del usuario? Si las tres respuestas son si, vas perfecto. Si alguna es no, regresa a los ejemplos de esa seccion y reescribelos a mano: la memoria muscular se construye escribiendo, no leyendo.

Un ultimo consejo de colega: guarda todos los programas de esta unidad en una carpeta de tu computadora. En unas semanas, cuando estes peleando con hashes, clases u otro tema, vas a querer un ejemplo de `each_with_index` o de un menu con `while true` a la mano, y nada mejor que tu propio codigo comentado por ti. Los buenos programadores no memorizan todo: construyen su propia biblioteca de ejemplos y la consultan sin verguenza. Esta unidad acaba de poner los primeros veinte libros en tu estante. A seguirle.

UNIDAD 5: Hashes y métodos

Si llegaste hasta aquí ya dominas los arreglos, los strings y las estructuras de control, y eso es bastante. Pero déjame decirte algo de colega a colega: con lo que sabes hasta ahora puedes hacer programas que funcionan, pero todavía no puedes hacer programas que se sientan profesionales. Te falta la pieza que conecta los datos con el significado, y te falta la pieza que convierte tu código en algo reutilizable. Esas dos piezas son los hashes y los métodos, y en esta unidad las vas a dominar a fondo.

Un hash es como un arreglo, pero en lugar de buscar elementos por su posición (el cero, el uno, el dos), los buscas por su nombre. Piensa en la diferencia entre decir "el elemento número 2 de la lista" y decir "el teléfono de María". Lo segundo es más natural, más legible y mucho menos propenso a errores. Cuando tu programa crezca y tengas que manejar usuarios, productos, configuraciones o cualquier cosa con atributos, vas a vivir dentro de los hashes. No es exageración: en el mundo real de Ruby, desde Rails hasta los scripts más sencillos, los hashes están en todas partes.

Los métodos, por otro lado, son la forma de empaquetar comportamiento. Hasta ahora tu código probablemente es una secuencia larga de instrucciones de arriba hacia abajo. Eso funciona para programas de 20 líneas, pero cuando llegues a 200 líneas se vuelve un desastre imposible de mantener. Los métodos te permiten decirle a Ruby: "mira, esta tarea que repito tanto, guárdala con este nombre, y cuando la necesite te la pido". Es como enseñarle un truco nuevo a un perro: lo entrenas una vez y después solo dices la palabra mágica.

En esta unidad vamos a ir paso por paso. Primero los hashes: qué son, las dos sintaxis para crearlos (el hash rocket y la sintaxis de símbolos), cómo leer, cambiar y borrar datos, cómo recorrerlos con `each`, y los métodos más útiles como `keys`, `values`, `merge` y `fetch`. Después pasamos a los métodos: cómo definirlos con `def`, cómo pasarles parámetros, el famoso retorno implícito de Ruby (una de las cosas que más confunden a los que vienen de otros lenguajes), parámetros por defecto, el operador `splat` para recibir argumentos variables, los argumentos de palabra clave, y las convenciones de nombres con signos de interrogación y admiración que hacen que el código Ruby se lea casi como español.

Prepárate porque esta unidad es larga, pero cada sección construye sobre la anterior. Al final vas a escribir programas que se ven y se sienten como código Ruby de verdad, no como traducciones de otro lenguaje.

Explicación detallada

Qué es un hash: el diccionario de Ruby

Imagina que tienes una caja con compartimentos numerados: el compartimento 0, el 1, el 2. Eso es un arreglo. Ahora imagina que en lugar de números, cada compartimento tiene una etiqueta con un nombre: "nombre", "edad", "ciudad". Eso es un hash. Un hash es una colección de pares clave-valor: cada valor vive bajo una clave que lo identifica, y para obtener el valor preguntas por la clave.

La analogía más clara es un diccionario de verdad: buscas una palabra (la clave) y obtienes su definición (el valor). O tu agenda de contactos: buscas "mamá" (la clave) y obtienes su número de teléfono (el valor). Nadie busca a su mamá diciendo "el contacto número 4". Usas el nombre porque el nombre tiene significado.

En Ruby, un hash se escribe con llaves y pares separados por comas:

```
persona = { "nombre" => "Carlos", "edad" => 30, "ciudad" => "Guadalajara" }
```

Aquí las claves son "nombre", "edad" y "ciudad", y los valores son "Carlos", 30 y "Guadalajara". Para obtener la edad escribes `persona["edad"]` y Ruby te devuelve 30. Fíjate en algo importante: los valores pueden ser de cualquier tipo (strings, números, arreglos, hasta otros hashes), y las claves también pueden ser de varios tipos, aunque lo más común es usar strings o símbolos (de los símbolos hablamos en la siguiente sección).

Los hashes brillan cuando los datos tienen estructura pero no tienen un orden natural. Un usuario tiene nombre, email y edad, pero no tiene sentido decir que el nombre es "el elemento 0". Un producto tiene precio, stock y categoría. Una configuración tiene opciones con nombres. En todos estos casos el hash es la herramienta correcta.

Otro punto clave: los hashes no garantizan un orden significativo como los arreglos. En Ruby moderno los hashes sí conservan el orden en que insertaste los pares, pero no debes depender de eso para tu lógica. Si el orden importa, usa un arreglo. Si lo que importa es encontrar cosas por su nombre, usa un hash.

Un error común de principiantes es usar arreglos para todo, incluso cuando los datos tienen nombres naturales. Mira la diferencia:

```
# Código malo: ¿qué significa cada posición?
datos = ["Ana", 25, "Monterrey"]
puts datos[1] # ¿1 es la edad? ¿o es otra cosa? Hay que adivinar.

# Código bueno: cada dato tiene nombre
datos = { "nombre" => "Ana", "edad" => 25, "ciudad" => "Monterrey" }
puts datos["edad"] # Clarísimo: quiero la edad.
```

El código bueno se explica solo. Dentro de seis meses, cuando vuelvas a leer tu programa, vas a agradecer haber usado hashes.

Hash rocket vs sintaxis de símbolos: las dos formas de escribir un hash

En Ruby existen dos sintaxis para crear hashes, y verás las dos en código real, así que necesitas manejarlas con soltura. La primera es la sintaxis del hash rocket, que usa => entre la clave y el valor:

```
config = { "host" => "localhost", "puerto" => 3000, "debug" => true }
```

Esta sintaxis funciona con cualquier tipo de clave: strings, números, símbolos, lo que sea. Es la sintaxis original, la de toda la vida, y sigue siendo perfectamente válida.

La segunda es la sintaxis de símbolos, que es un atajo cuando las claves son símbolos:

```
config = { host: "localhost", puerto: 3000, debug: true }
```

Fíjate bien: host: es lo mismo que :host =>. Ruby lo convierte internamente al mismo hash. Esta sintaxis es más corta, más limpia y es la que se usa en el código moderno, especialmente en Rails y en las gemas actuales.

Ahora, ¿qué es un símbolo? Un símbolo es como un string ligero e inmutable que se usa como etiqueta o identificador. Se escribe con dos puntos al inicio: :nombre, :edad, :ciudad. La diferencia técnica con un string es que cada vez que escribes "nombre" Ruby crea un objeto string nuevo en memoria, mientras que :nombre es siempre el mismo objeto. Para claves de hash, que se repiten miles de veces, los símbolos son más eficientes y más idiomáticos.

Mira la comparación lado a lado:

```
# Hash rocket con strings
usuario1 = { "nombre" => "Luis", "email" => "luis@ejemplo.com" }

# Hash rocket con símbolos (sintaxis vieja pero válida)
usuario2 = { :nombre => "Luis", :email => "luis@ejemplo.com" }

# Sintaxis de símbolos (moderna, preferida)
usuario3 = { nombre: "Luis", email: "luis@ejemplo.com" }

# Las tres son equivalentes en contenido. La tercera es la más rubyista.
```

¿Cuándo usar cada una? Regla práctica: si las claves son símbolos (que será el 90 por ciento de las veces), usa la sintaxis moderna { nombre: "Luis" }. Si las claves son strings, números u otra cosa, usa el hash rocket. Y si estás leyendo código viejo, no te asustes al ver :nombre => "Luis": es exactamente lo mismo que nombre: "Luis".

Un detalle que confunde a muchos: con la sintaxis de símbolos NO puedes usar claves que no sean símbolos. Esto es un error:

```
# Código malo: esto ni siquiera es sintaxis válida
# datos = { "nombre": "Ana" } # SyntaxError
```

Si la clave es un string, forzosamente usas el hash rocket: { "nombre" => "Ana" }. No hay atajo para eso.

Otro error típico es mezclar sin querer claves símbolo y claves string para lo mismo. Mira:

```
# Código malo: dos claves distintas sin darte cuenta
usuario = { nombre: "Pedro" }
puts usuario["nombre"] # nil. ¡La clave es :nombre, no "nombre"!

# Código bueno: sé consistente
usuario = { nombre: "Pedro" }
puts usuario[:nombre] # "Pedro". La clave símbolo se busca con símbolo.
```

Esto es importantísimo: :nombre y "nombre" son claves DIFERENTES para Ruby. Un hash puede tener las dos al mismo tiempo y son dos entradas separadas. Cuando accedas a un hash, usa el mismo tipo de clave con el que lo creaste. Si usas la sintaxis moderna, accede con símbolos. Si usas hash rocket con strings, accede con strings.

Acceder, modificar y eliminar pares clave-valor

Ya sabes crear un hash. Ahora lo esencial del día a día: leer valores, cambiarlos, agregar pares nuevos y borrar los que sobren.

Para leer un valor usas corchetes con la clave:

```
alumno = { nombre: "Sofía", edad: 22, carrera: "Sistemas" }
puts alumno[:nombre]    # "Sofía"
puts alumno[:edad]      # 22
puts alumno[:carrera]   # "Sistemas"
```

¿Qué pasa si pides una clave que no existe? Ruby no se enoja ni lanza error: te devuelve nil. Esto es una arma de doble filo. Por un lado es cómodo; por otro, si escribes mal la clave, tu programa sigue corriendo con nil y el error aparece mucho después, lejos de donde lo causaste. Más adelante veremos fetch, que es la versión estricta.

Para modificar un valor existente o agregar un par nuevo, la sintaxis es idéntica: asignas con corchetes.

```
alumno = { nombre: "Sofía", edad: 22 }
alumno[:edad] = 23          # Modifica: la clave ya existía
alumno[:ciudad] = "Puebla"  # Agrega: la clave no existía, se crea
puts alumno.inspect
# {:nombre=>"Sofía", :edad=>23, :ciudad=>"Puebla"}
```

Fíjate que no hay diferencia entre "modificar" y "agregar": si la clave existe se actualiza el valor, si no existe se crea el par. Esto es práctico pero exige cuidado: si escribes mal una clave al intentar modificarla, en lugar de un error obtienes un par nuevo basura.

```
# Código malo: typo silencioso
precios = { manzana: 10, pera: 12 }
precios[:mansana] = 15  # Querías actualizar :manzana, creaste :mansana
puts precios.inspect
# {:manzana=>10, :pera=>12, :mansana=>15} # Basura silenciosa

# Código bueno: verifica antes si la clave debe existir
precios = { manzana: 10, pera: 12 }
if precios.key?(:manzana)
  precios[:manzana] = 15
end
```

Para eliminar un par usas el método delete, que recibe la clave y devuelve el valor eliminado:

```
inventario = { laptop: 5, mouse: 20, teclado: 15 }  
borrado = inventario.delete(:mouse)  
puts borrado           # 20 (el valor que tenía)  
puts inventario.inspect # {:laptop=>5, :teclado=>15}
```

Si la clave no existe, delete devuelve nil sin hacer nada. También puedes vaciar todo el hash con clear, aunque eso se usa poco.

Hay dos métodos de consulta que vas a usar muchísimo: key? (también escrito has_key?) te dice si una clave existe, y value? te dice si un valor existe.

```
usuario = { nombre: "Elena", rol: "admin" }  
puts usuario.key?(:nombre) # true  
puts usuario.key?(:edad)   # false  
puts usuario.value?("admin") # true
```

Acostúmbrate a usar key? antes de leer claves que podrían no existir, o mejor aún, usa fetch, que veremos en la sección de métodos útiles. La idea es: nunca confíes ciegamente en que una clave existe, especialmente si el hash viene de fuera de tu programa (un formulario, una API, un archivo).

Recorrer hashes con each: claves y valores de paseo

Igual que los arreglos, los hashes se pueden recorrer con each. La diferencia es que cada vuelta te entrega DOS cosas: la clave y el valor. Por eso el bloque de each para hashes recibe dos parámetros separados por coma entre plecas:

```
precios = { manzana: 10, pera: 12, uva: 18 }  
precios.each do |fruta, precio|  
  puts "La #{fruta} cuesta ${precio}"  
end  
# La manzana cuesta $10  
# La pera cuesta $12  
# La uva cuesta $18
```

Fíjate en |fruta, precio|: el primer parámetro siempre recibe la clave y el segundo el valor. Los nombres los eliges tú; lo importante es el orden. Ponles nombres descriptivos porque se leen en cada vuelta.

Dentro del bloque puedes hacer lo que sea: filtrar, transformar, acumular. Un patrón muy común es construir un hash nuevo a partir de otro:

```
# Aplicar 10% de descuento y guardar en un hash nuevo
precios = { manzana: 10, pera: 12, uva: 18 }
con_descuento = {}
precios.each do |fruta, precio|
  con_descuento[fruta] = precio * 0.9
end
puts con_descuento.inspect
# {:manzana=>9.0, :pera=>10.8, :uva=>16.2}
```

También puedes recorrer solo las claves con `each_key` o solo los valores con `each_value`, cuando no necesitas la otra mitad:

```
usuario = { nombre: "Raúl", edad: 35, ciudad: "Tijuana" }
usuario.each_key do |clave|
  puts "Campo: #{clave}"
end
usuario.each_value do |valor|
  puts "Valor: #{valor}"
end
```

Un error común es tratar al bloque de `each` como si recibiera un solo parámetro y luego intentar usarlo como hash:

```
# Código malo: cada vuelta entrega un arreglo [clave, valor], no un hash
precios = { manzana: 10, pera: 12 }
precios.each do |par|
  puts par[:manzana] # NoMethodError: estás indexando un arreglo con símbolo
end

# Código bueno: recibe clave y valor por separado
precios.each do |fruta, precio|
  puts "#{fruta}: #{precio}"
end
```

Si por alguna razón recibes un solo parámetro, Ruby te entrega cada par como un arreglo de dos elementos `[clave, valor]`. Puedes desarmarlo, pero lo normal y legible es usar dos parámetros.

Otro patrón útil: `each` con índice no existe directamente en hashes como en arreglos, pero rara vez lo necesitas porque las claves ya te dan identidad. Si de verdad necesitas un contador, lleva tu propia variable o usa `each_with_index`, que te da el par y el índice.

Métodos útiles de hash: keys, values, merge, fetch y compañía

Los hashes traen una caja de herramientas muy completa. Estos son los métodos que vas a usar el 95 por ciento del tiempo; conócelos bien y te van a ahorrar muchísimas líneas.

keys te devuelve un arreglo con todas las claves, y values un arreglo con todos los valores:

```
curso = { titulo: "Ruby básico", duracion: 40, nivel: "inicial" }
puts curso.keys.inspect    # [:titulo, :duracion, :nivel]
puts curso.values.inspect  # ["Ruby básico", 40, "inicial"]
```

Esto es útil cuando necesitas procesar las claves o los valores con métodos de arreglo (map, select, sort, etc.). Por ejemplo, verificar que un hash tenga todas las claves obligatorias:

```
obligatorias = [:titulo, :duracion, :nivel]
faltantes = obligatorias - curso.keys
puts "Faltan: #{faltantes.inspect}"
```

merge combina dos hashes en uno nuevo. Si hay claves repetidas, ganan los valores del hash que pasas como argumento:

```
base = { nombre: "Ana", edad: 28 }
extra = { ciudad: "León", edad: 29 }
combinado = base.merge(extra)
puts combinado.inspect
# {:nombre=>"Ana", :edad=>29, :ciudad=>"León"}
puts base.inspect
# {:nombre=>"Ana", :edad=>28} # base NO cambió
```

Ojo con el detalle: merge NO modifica el hash original, devuelve uno nuevo. Si quieres modificar el original, usa merge! con signo de admiración (hablaremos de la convención de ! más adelante, pero adelanto: el ! suele significar "versión que modifica el objeto").

fetch es la forma segura y estricta de leer un valor. A diferencia de los corchetes, que devuelven nil si la clave no existe, fetch te permite decidir qué pasa:

```

config = { host: "localhost", puerto: 3000 }

# Con corchetes: nil silencioso
puts config[:usuario] # nil (y ni te avisa)

# Con fetch y valor por defecto
puts config.fetch(:usuario, "invitado") # "invitado"

# Con fetch estricto: lanza error si falta la clave
puts config.fetch(:password) # KeyError: key not found: :password

```

¿Cuándo usar cada uno? Usa corchetes cuando la clave es opcional y nil es una respuesta aceptable. Usa fetch con valor por defecto cuando quieras un plan B explícito. Usa fetch sin valor por defecto cuando la clave DEBE existir y prefieres que el programa truene de inmediato con un error claro en lugar de seguir con nil y fallar misteriosamente diez líneas después. Los errores tempranos y ruidosos son tus amigos; los nil silenciosos son tus enemigos.

Otros métodos que vale la pena conocer: length o size te dicen cuántos pares hay; empty? te dice si el hash está vacío; invert intercambia claves por valores (cuidado si hay valores repetidos, se pierden); to_a convierte el hash en un arreglo de pares; y slice (en Ruby moderno) te devuelve un hash solo con las claves que pidas:

```

usuario = { nombre: "Jorge", edad: 31, ciudad: "Mérida", rol: "editor" }
puts usuario.length # 4
puts usuario.empty? # false
puts usuario.slice(:nombre, :rol).inspect
# {:nombre=>"Jorge", :rol=>"editor"}

```

También existe transform_values, que aplica un bloque a cada valor y devuelve un hash nuevo. Es elegantísimo para ajustes en masa:

```

precios = { manzana: 10, pera: 12, uva: 18 }
iva = precios.transform_values { |p| p * 1.16 }
puts iva.inspect
# {:manzana=>11.6, :pera=>13.92, :uva=>20.88}

```

Errores comunes con hashes: la galería de los horrores

Vamos a juntar los errores típicos en un solo lugar porque reconocerlos te va a ahorrar horas de depuración.

Error 1: confundir símbolos con strings como claves. Ya lo vimos, pero lo repito porque es el error número uno: { nombre: "Ana" } se accede con [:nombre], no con ["nombre"]. Si vienes de JavaScript, donde objeto.clave y objeto["clave"] son lo mismo, esto te va a morder. En Ruby no son lo mismo.

Error 2: asumir que una clave existe y operar con nil sin darte cuenta.

```
# Código malo
usuarios = { ana: 25, luis: 30 }
edad = usuarios[:pedro] + 1 # NoMethodError: nil + 1 no existe

# Código bueno
edad = usuarios.fetch(:pedro, 0) + 1 # 1, sin true
```

Error 3: modificar un hash mientras lo recorres con each. Agregar o borrar claves durante la iteración puede dar resultados raros o errores. Si necesitas filtrar, construye un hash nuevo o usa select/reject, que devuelven hashes nuevos:

```
# Código malo
precios = { manzana: 10, pera: 12, uva: 18 }
precios.each do |fruta, precio|
  precios.delete(fruta) if precio > 11 # Peligroso: mutar durante each
end

# Código bueno
precios = { manzana: 10, pera: 12, uva: 18 }
baratos = precios.select { |fruta, precio| precio <= 11 }
puts baratos.inspect # {:manzana=>10}
```

Error 4: usar hashes cuando un objeto sería mejor. Si tu hash empieza a tener "métodos" imaginarios (varias partes del código hacen lo mismo con los mismos datos), es señal de que necesitas una clase. Los hashes guardan datos; las clases guardan datos CON comportamiento. De eso trata la Unidad 6.

Error 5: anidar hashes sin control hasta crear un laberinto.

```
# Código malo: ¿quién se acuerda de esta ruta?
valor = datos[:a][:b][:c][:d]

# Código bueno: usa dig, que devuelve nil si algo falta en el camino
valor = datos.dig(:a, :b, :c, :d)
```

dig es un salvavidas para hashes anidados: recorre la ruta y si cualquier nivel es nil, devuelve nil en lugar de tronar con NoMethodError.

Definir métodos con def: tu primera fábrica de comportamiento

Hasta aquí todo tu código ha sido una lista de instrucciones que Ruby ejecuta de arriba hacia abajo. Un método es un bloque de código con nombre que puedes ejecutar cuando quieras, cuantas veces quieras. Se define con def, se termina con end, y se usa escribiendo su nombre:

```
def saludar
  puts "Hola, ¿cómo estás?"
end

saludar # Llama al método: imprime el saludo
saludar # Lo puedes llamar cuantas veces quieras
```

La analogía: un método es como una receta de cocina. La escribes una vez (def), y cada vez que quieres el platillo sigues la receta (llamas al método). Sin métodos, sería como tener que inventar la receta desde cero cada vez que tienes hambre.

Los métodos de verdad despegan cuando reciben parámetros: valores que les pasas al llamarlos para que trabajen con ellos. Los parámetros se declaran entre el nombre y el end, y los argumentos son los valores concretos que envías en la llamada:

```
def saludar(nombre)
  puts "Hola, #{nombre}, ¿cómo estás?"
end

saludar("María") # Hola, María, ¿cómo estás?
saludar("Pedro") # Hola, Pedro, ¿cómo estás?
```

Terminología que no debes confundir: parámetro es la variable en la definición (nombre), argumento es el valor en la llamada ("María"). Es como la diferencia entre el molde y la galleta: el parámetro es el espacio vacío en el molde, el argumento es la masa que le pones.

Un método puede recibir varios parámetros, separados por comas. En la llamada debes pasar el mismo número de argumentos, en el mismo orden. Estos se llaman parámetros posicionales porque la posición es lo que importa:

```
def presentar(nombre, edad, ciudad)
  puts "#{nombre} tiene #{edad} años y vive en #{ciudad}."
end

presentar("Lucía", 27, "Oaxaca")
# Lucía tiene 27 años y vive en Oaxaca.

# Código malo: orden equivocado, resultado absurdo pero sin error
presentar("Oaxaca", "Lucía", 27)
# Oaxaca tiene Lucía años y vive en 27. ¡Ruby no se queja!
```

Fíjate en el peligro: Ruby no sabe que "Oaxaca" no es un nombre. Los parámetros posicionales son ciegos al significado; solo cuentan posiciones. Por eso, cuando un método recibe más de dos o tres parámetros, conviene usar argumentos de palabra clave (los veremos más adelante): `presentar(nombre: "Lucía", edad: 27, ciudad: "Oaxaca")` no se puede equivocar de orden.

Si llamas a un método con menos argumentos de los que declara, Ruby lanza `ArgumentError`. Si pasas más, también. El número debe cuadrar exacto, salvo que uses parámetros por defecto o `splat`, que veremos enseguida.

Buena práctica desde el día uno: un método debe hacer UNA sola cosa y su nombre debe decir qué cosa. `saludar` saluda. `calcular_promedio` calcula un promedio. Si tu método se llama `procesar_datos_y_enviar_correo_y_actualizar_base`, en realidad son tres métodos disfrazados de uno. Divídelo.

Otra buena práctica: los nombres de métodos en Ruby van en minúsculas con palabras separadas por guion bajo (`snake_case`): `calcular_total`, `buscar_usuario`, `validar_email`. Nada de `camelCase`; eso es de otros lenguajes.

El retorno implícito: la última línea manda

En muchos lenguajes tienes que escribir `return` para devolver un valor desde una función. En Ruby el `return` existe, pero casi nunca lo necesitas, porque Ruby devuelve automáticamente el valor de la ÚLTIMA línea ejecutada en el método. Esto se llama retorno implícito y es una de las señas de identidad del lenguaje:

```
def sumar(a, b)
  a + b
end

resultado = sumar(3, 4)
puts resultado # 7
```

No hay return por ningún lado, y sin embargo sumar(3, 4) vale 7. La última expresión, `a + b`, se convierte en el valor de retorno. Esto aplica a todo: la última línea puede ser un string, un arreglo, un hash, un cálculo, lo que sea.

```
def clasificar(edad)
  if edad >= 18
    "adulto"
  else
    "menor"
  end
end

puts clasificar(20) # "adulto"
puts clasificar(15) # "menor"
```

Aquí la última expresión es el `if` completo, y el `if` vale lo que valga la rama ejecutada. Todo en Ruby es una expresión que produce un valor; no hay "instrucciones vacías".

Entonces, ¿para qué sirve `return`? Para salir ANTES de tiempo, cuando ya tienes la respuesta y no quieres seguir ejecutando el resto:

```
def buscar_positivo(numeros)
  numeros.each do |n|
    return n if n > 0 # Encontré uno: me voy, no sigo buscando
  end
  nil # Si llegué aquí, no había ningún positivo
end

puts buscar_positivo([-3, -1, 5, 8]) # 5
puts buscar_positivo([-3, -1]).inspect # nil
```

El `return` explícito es para retornos tempranos (early return), un patrón muy útil para manejar casos especiales al inicio y dejar el flujo principal limpio:

```
def dividir(a, b)
  return "No se puede dividir entre cero" if b == 0
  a / b.to_f
end

puts dividir(10, 2) # 5.0
puts dividir(10, 0) # No se puede dividir entre cero
```

Sin el return temprano, tendrías que anidar todo en un if/else. Con el return temprano, el caso raro se atiende y se despacha en la primera línea, y el resto del método se lee de corrido.

Error común de principiantes: poner puts como última línea creyendo que eso "devuelve" el valor.

```
# Código malo: puts devuelve nil, no el texto
def doble(n)
  puts n * 2
end
x = doble(5) # Imprime 10 en pantalla...
puts x.inspect # ...pero x es nil

# Código bueno: la última línea es el valor
def doble(n)
  n * 2
end
x = doble(5)
puts x # 7... digo, 10. El valor quedó guardado en x.
```

Regla de oro: puts es para mostrarle cosas al humano; el valor de retorno es para entregarle cosas al programa. Un método que calcula algo debe devolver el valor, no imprimirlo. Si lo imprime, el resto del programa no puede usar el resultado. Decide: ¿este método informa (puts) o produce (retorna)? Lo ideal es que produzca, y que sea quien lo llama quien decida si imprimirlo.

Parámetros por defecto: valores cuando el usuario no dice nada

Hay métodos donde casi siempre usas el mismo valor para un parámetro, y solo de vez en cuando quieres cambiarlo. Para esos casos existen los parámetros por defecto: si en la llamada no pasas ese argumento, Ruby usa el valor que declaraste.

```
def saludar(nombre, saludo = "Hola")
  puts "#{saludo}, #{nombre}."
end

saludar("Ana")           # Hola, Ana. (usó el valor por defecto)
saludar("Ana", "Buenos días") # Buenos días, Ana. (lo sobrescribió)
```

La analogía: es como pedir tacos. "¿De qué los quiere?" "De pastor" es tu valor por defecto: si no dices nada, te dan de pastor. Pero siempre puedes pedir de bistec. El parámetro por defecto es tu "lo de siempre".

Los parámetros con valor por defecto van DESPUÉS de los obligatorios. Esto tiene lógica: si alguien llama con un solo argumento, Ruby debe saber a qué parámetro corresponde (al primero, el obligatorio).

```
# Código bueno
def conectar(host, puerto = 80, timeout = 30)
  puts "Conectando a #{host}:#{puerto} (timeout #{timeout}s)"
end
conectar("ejemplo.com")           # usa 80 y 30
conectar("ejemplo.com", 8080)     # usa 8080 y 30
conectar("ejemplo.com", 8080, 10) # usa los tres dados

# Código malo: defecto antes que obligatorio (SyntaxError)
# def conectar(puerto = 80, host)
# end
```

Un detalle fino: el valor por defecto se evalúa en cada llamada, no una sola vez al definir el método. Esto importa si el valor por defecto es algo mutable como un arreglo o un hash: cada llamada recibe un objeto fresco, así que no hay sorpresas de estado compartido (en Python esto es una trampa famosa; en Ruby no).

```
def agregar(item, lista = [])
  lista << item
  lista
end

puts agregar("a").inspect # ["a"]
puts agregar("b").inspect # ["b"], NO ["a", "b"]. Cada llamada tiene su lista.
```

Los parámetros por defecto hacen tus métodos más cómodos sin romper a quienes ya los usan: puedes agregar un parámetro nuevo con valor por defecto a un método existente y todo el código viejo sigue funcionando. Es la forma elegante de evolucionar una API.

El operador splat: *args para recibir lo que sea

¿Qué pasa cuando no sabes cuántos argumentos te van a pasar? Por ejemplo, un método que suma cualquier cantidad de números: a veces le pasan dos, a veces diez. Para eso existe el operador splat: un asterisco antes del parámetro que recoge todos los argumentos sobrantes en un arreglo.

```
def sumar(*numeros)
  total = 0
  numeros.each { |n| total += n }
  total
end

puts sumar(1, 2)          # 3
puts sumar(1, 2, 3, 4)    # 10
puts sumar()              # 0 (arreglo vacío, sin trueno)
```

La analogía: *args es como una bolsa. Los parámetros normales son vasos medidos: cada uno recibe exactamente una porción. El splat es la bolsa donde cae todo lo que sobra, y dentro del método la tratas como un arreglo normal con each, length, sum, etc.

Puedes combinar parámetros normales con splat: los primeros argumentos llenan los parámetros normales (en orden) y el resto cae en la bolsa.

```
def presentar(lider, *equipo)
  puts "Líder: #{lider}"
  puts "Equipo: #{equipo.join(", ")}"
end

presentar("Ana", "Luis", "Pedro", "Sofía")
# Líder: Ana
# Equipo: Luis, Pedro, Sofía
```

El splat también funciona al revés: si tienes un arreglo y quieres pasarlo como argumentos separados, pones * delante en la llamada. Esto se llama "desempaquetar".

```
def sumar(a, b, c)
  a + b + c
end

numeros = [1, 2, 3]
puts sumar(*numeros) # 6. El arreglo se desempaquetó en 3 argumentos.
```

Y hay un truco clásico: convertir argumentos en arreglo y viceversa sin esfuerzo. El `splat` es una de esas herramientas que al principio parece magia y después no puedes vivir sin ella.

Error común: poner el `splat` en medio y confundirse con qué recibe cada parámetro. Funciona (Ruby reparte con inteligencia), pero oscurece el código. Mejor práctica: el `splat` va al final de la lista de parámetros, después de los obligatorios y los que tienen valor por defecto.

```
# Válido pero confuso
def raro(a, *medio, z)
  puts "a=#{a}, medio=#{medio.inspect}, z=#{z}"
end
raro(1, 2, 3, 4) # a=1, medio=[2, 3], z=4

# Mejor: splat al final, todo claro
def claro(a, z, *resto)
  puts "a=#{a}, z=#{z}, resto=#{resto.inspect}"
end
```

Argumentos de palabra clave: adiós al desorden posicional

Recuerda el problema de `presentar("Oaxaca", "Lucía", 27)`: con parámetros posicionales, si te equivocas de orden, Ruby no se entera y el resultado es un absurdo silencioso. Los argumentos de palabra clave resuelven esto: cada argumento lleva su nombre, así que el orden deja de importar y el código se lee solo.

```
def presentar(nombre:, edad:, ciudad:)
  puts "#{nombre} tiene #{edad} años y vive en #{ciudad}."
end

presentar(nombre: "Lucía", edad: 27, ciudad: "Oaxaca") # perfecto
presentar(ciudad: "Oaxaca", nombre: "Lucía", edad: 27) # también perfecto
presentar("Lucía", 27, "Oaxaca") # ArgumentError: ya no acepta posicionales
```


Fíjate en la sintaxis: en la definición, nombre: (con dos puntos DESPUÉS) marca un argumento de palabra clave obligatorio. En la llamada se ve igual que un hash, y de hecho Ruby los trata de forma muy parecida por dentro. La diferencia es que aquí los nombres están declarados y validados.

Si omites un argumento de palabra clave obligatorio, Ruby truena con `ArgumentError` diciendo cuál faltó. Eso es mil veces mejor que un nil silencioso:

```
presentar(nombre: "Lucía", edad: 27)
# ArgumentError: missing keyword: :ciudad
```

También puedes darles valores por defecto, igual que a los posicionales:

```
def conectar(host:, puerto: 80, timeout: 30)
  puts "Conectando a #{host}:#{puerto} (timeout #{timeout}s)"
end

conectar(host: "ejemplo.com")           # 80 y 30 por defecto
conectar(host: "ejemplo.com", puerto: 8080) # timeout 30 por defecto
```

Y existe el doble splat **opciones**, que recoge todas las palabras clave no declaradas en un hash. Es el primo del `*args` pero para keywords:

```
def configurar(**opciones)
  puts opciones.inspect
end

configurar(color: "rojo", tamano: 12, activo: true)
# {:color=>"rojo", :tamano=>12, :activo=>true}
```

¿Cuándo usar posicionales y cuándo palabras clave? Regla práctica de colega: si el método recibe 1 o 2 parámetros obvios (como `sumar(a, b)` o `saludar(nombre)`), los posicionales están bien. Si recibe 3 o más, o si alguno es opcional/confundible (¿qué significa `conectar("x", 8080, 10)?`), usa palabras clave. El código `presentar(nombre: "Lucía", edad: 27, ciudad: "Oaxaca")` no necesita comentarios: se explica solo. Esa legibilidad es oro cuando otro programador (o tú dentro de tres meses) lea tu código.

Un error típico es mezclar estilos sin querer al llamar:

```
# Código malo: la definición pide keywords, la llamada da posicionales
def crear_usuario(nombre:, email:)
  puts "#{nombre} <#{email}>"
end

# crear_usuario("Ana", "ana@mail.com") # ArgumentError

# Código bueno: respeta el estilo declarado
crear_usuario(nombre: "Ana", email: "ana@mail.com")
```

Métodos con ? y !: la convención que hace Ruby legible

En la mayoría de los lenguajes, los nombres de métodos solo pueden tener letras, números y guion bajo. Ruby permite dos signos extra al final del nombre: ? y !. No son decoración: son convenciones con significado que toda la comunidad respeta.

El ? al final indica que el método responde una PREGUNTA y devuelve true o false (un predicado). Léelo como una pregunta en español:

```
numero = 7
puts numero.odd?    # true  ("¿7 es impar?")
puts numero.even?   # false ("¿7 es par?")

texto = ""
puts texto.empty?   # true  ("¿el texto está vacío?")

lista = [1, 2, 3]
puts lista.include?(2) # true ("¿la lista incluye al 2?")
```

Cuando definas tus propios métodos que respondan sí/no, ponles ?: es_par?(n), mayor_de_edad?(edad), email_valido?(correo). El que lea tu código va a entender al instante que eso devuelve un booleano.

El ! al final indica PELIGRO o MUTACIÓN: el método modifica el objeto sobre el que se llama, en lugar de devolver una copia nueva. Es la versión "agresiva" de un método que también existe sin !:

```

texto = "hola"
mayus = texto.upcase    # Devuelve "HOLA", pero texto sigue siendo "hola"
puts texto              # "hola" (intacto)

texto.upcase!           # Modifica el objeto original
puts texto              # "HOLA" (cambió para siempre)

```

La pareja método/método! es un patrón constante en Ruby: sort vs sort!, merge vs merge!, gsub vs gsub!, strip vs strip!. La versión sin ! es la segura (devuelve algo nuevo, no toca el original); la versión con ! es la que muta.

Pero ojo, y esto es importante: el ! NO garantiza nada por sí solo. Es una convención, no una regla del lenguaje. Ruby no te impide definir un método con ! que no mute nada, ni uno sin ! que sí mute. La convención solo funciona si todos la respetamos. Cuando defines métodos destructivos, ponles !. Cuando defines la versión segura, no se lo pongas. Y nunca uses ! solo "para que se vea rudo": si tu método no muta nada, el ! es una mentira que va a confundir a quien lo use.

Otro uso legítimo del !: métodos que lanzan una excepción cuando algo sale mal, frente a una versión sin ! que devuelve nil o false. El ejemplo clásico es save vs save! en Rails: save devuelve false si falla, save! truena con excepción. La idea general del ! es "esta versión es más estricta / más peligrosa".

```

# Convención bien aplicada
def activar!
  @activo = true    # muta el estado interno: el ! avisa
end

def activo?
  @activo           # pregunta: el ? avisa que devuelve booleano
end

```

Buenas prácticas para definir métodos

Cierra la sección de métodos con hábitos que separan al principiante del programador serio.

Primero: un método, una responsabilidad. Si para describir lo que hace necesitas la palabra "y", probablemente son dos métodos. calcular_total_y_aplicar_descuento_y_guardar es tres métodos con un solo nombre. Sepáralos y cada uno será más fácil de probar, de entender y de reutilizar.

Segundo: nombres que dicen qué hacen, no cómo lo hacen. `buscar_usuario_por_email` es mejor que `query_db_users_email`. Y evita nombres vagos como `procesar`, `manejar`, `hacer_cosas`: no dicen nada. Si no puedes nombrar bien un método, quizá todavía no entiendes bien qué hace; aclara la idea y el nombre sale solo.

Tercero: métodos cortos. No hay una ley escrita, pero si tu método pasa de 15 o 20 líneas, casi seguro está haciendo de más. Los métodos cortos se leen de un vistazo, se prueban fácil y se reutilizan más.

Cuarto: devuelve valores, no imprimas. Ya lo vimos: un método que calcula debe retornar el resultado y dejar que quien lo llama decida qué hacer con él (imprimirlo, guardarlo, pasarlo a otro método). Reserva puts para programas pequeños, depuración o métodos cuya única misión sea mostrar algo en pantalla.

Quinto: prefiere pocos parámetros. Cada parámetro es una cosa más que recordar al llamar el método. Si necesitas más de tres o cuatro, considera palabras clave o agrupa datos relacionados en un hash o un objeto. Un método con siete parámetros posicionales es una invitación al error.

Sexto: no repitas lógica; si copias y pegas el mismo bloque de código en dos lugares, ese bloque quiere ser un método. La duplicación es deuda: cuando haya que cambiar la lógica, tendrás que acordarte de cambiarla en todos los lugares copiados, y te vas a olvidar de alguno. Un método centraliza el cambio en un solo lugar.

Hashes anidados y estructuras de datos del mundo real

En los ejemplos hasta ahora los valores de los hashes han sido simples: strings, números, booleanos. Pero un valor puede ser CUALQUIER cosa, incluyendo otro hash o un arreglo. Cuando metes hashes dentro de hashes (o arreglos dentro de hashes), obtienes estructuras anidadas capaces de representar datos del mundo real: un usuario con su dirección, un pedido con su lista de productos, la respuesta de una API.

```

usuario = {
  nombre: "Elena",
  edad: 29,
  direccion: {
    calle: "Av. Juárez 123",
    ciudad: "Guadalajara",
    cp: "44100"
  },
  telefonos: ["555-1111", "555-2222"]
}

```

Aquí `:direccion` apunta a otro hash y `:telefonos` a un arreglo. Para llegar a la ciudad escribes `usuario[:direccion][:ciudad]`: primero entras al hash interno y luego pides su clave. Cada nivel de corchetes baja un piso en la estructura. Léelo de izquierda a derecha como una ruta: "del usuario, la dirección, la ciudad".

La analogía: es como una carpeta con subcarpetas. `usuario` es la carpeta principal; dentro hay un documento (`nombre`), otro documento (`edad`), una subcarpeta (`direccion`) con sus propios documentos, y una lista (`telefonos`). Para llegar a un documento profundo abres carpeta tras carpeta.

El patrón más común en el mundo real es el **ARREGLO DE HASHES**: una lista donde cada elemento es un hash con la misma forma. Así se representan tablas, catálogos, resultados de búsquedas:

```

productos = [
  { nombre: "Laptop", precio: 15000, stock: 5 },
  { nombre: "Mouse", precio: 300, stock: 50 },
  { nombre: "Teclado", precio: 800, stock: 20 }
]

productos.each do |p|
  puts "#{p[:nombre]}: ${p[:precio]} (#{p[:stock]} en stock)"
end

# Filtrar con select: productos caros
caros = productos.select { |p| p[:precio] > 1000 }
puts caros.length # 1

```

Fíjate qué natural se vuelve: el arreglo te da la colección, cada hash te da los atributos con nombre, y los métodos de enumerable (`each`, `select`, `map`) trabajan sobre ellos sin fricción. Esto es exactamente lo que te devuelve una API en formato JSON convertido a Ruby, o lo que

lees de una base de datos: arreglos de hashes. Dominar esta estructura es dominar el 80 por ciento del manejo de datos en Ruby.

El peligro de las estructuras anidadas es el nil intermedio. Si cualquier nivel de la ruta no existe, obtienes NoMethodError al intentar indexar nil:

```
# Código malo
datos = { usuario: { nombre: "Ana" } }
puts datos[:usuario][:direccion][:ciudad]
# NoMethodError: undefined method '[]' for nil
# (:direccion no existe, devuelve nil, y nil[:ciudad] truena)
```

Tres defensas, de manual a elegante. Primera: verificar cada nivel (verboso pero explícito). Segunda: usar dig, que recorre la ruta y devuelve nil si algo falta en el camino, sin tronar. Tercera: fetch encadenado con defaults. En la práctica, dig es tu mejor amigo:

```
# Código bueno: dig no truena
puts datos.dig(:usuario, :direccion, :ciudad).inspect # nil, sin error
puts usuario.dig(:direccion, :ciudad) # "Guadalajara" (cuando sí existe)
```

dig funciona también mezclando arreglos y hashes: datos.dig(:usuarios, 0, :nombre) entra al hash, toma el elemento 0 del arreglo y pide su :nombre. Es la navaja suiza para datos anidados que vienen de fuera (APIs, archivos, formularios), donde nunca puedes confiar en que la estructura esté completa.

Construir estructuras anidadas desde cero también es un patrón diario. Por ejemplo, agrupar una lista de hashes por una clave:

```
ventas = [
  { vendedor: "Ana", monto: 100 },
  { vendedor: "Luis", monto: 200 },
  { vendedor: "Ana", monto: 150 }
]

por_vendedor = {}
ventas.each do |v|
  nombre = v[:vendedor]
  por_vendedor[nombre] = [] unless por_vendedor.key?(nombre)
  por_vendedor[nombre] << v[:monto]
end
puts por_vendedor.inspect
# {"Ana"=>[100, 150], "Luis"=>[200]}
```

Creemos un hash donde cada clave es un vendedor y cada valor es un arreglo de montos. El `unless` con `key?` inicializa el arreglo la primera vez que aparece cada vendedor. (En Ruby existe `group_by` que hace esto en una línea, pero entender la versión manual te enseña la mecánica que `group_by` esconde.)

Buena práctica con anidados: no anides más de 3 niveles si puedes evitarlo. Cada nivel extra es un nivel más de corchetes que leer y un nivel más donde puede esconderse un `nil`. Si tu estructura tiene 5 niveles, probablemente necesitas objetos (Unidad 6) en lugar de hashes: un Pedido que TIENE productos es más claro que un hash de 5 pisos. Los hashes anidados son excelentes para datos; cuando esos datos empiezan a necesitar comportamiento, es hora de clases.

Ejemplos paso a paso

Ejemplo 1: agenda de contactos con hash. Este programa muestra lo básico: crear un hash con sintaxis de símbolos, agregar contactos, buscar por nombre y listar todo con `each`. Es el "hola mundo" de los hashes: si entiendes este programa, entiendes la mecánica esencial.

```
agenda = { ana: "555-1234", luis: "555-5678" }

# Agregar un contacto nuevo
agenda[:pedro] = "555-9012"

# Buscar un contacto
puts "El teléfono de Ana es: #{agenda[:ana]}"

# Listar todos con each
puts "--- Agenda completa ---"
agenda.each do |nombre, telefono|
  puts "#{nombre}: #{telefono}"
end
```

Fíjate en cómo cada operación usa la clave como identificador: `agenda[:ana]` lee, `agenda[:pedro] = ...` escribe. El `each` recibe `|nombre, telefono|` porque cada vuelta entrega el par completo. Este patrón (hash + `each` para listar) lo vas a repetir cientos de veces en tu carrera.

Ejemplo 2: convertir sintaxis y acceder con seguridad. Aquí practicamos las dos sintaxis de hash y el uso de `fetch` para evitar `nil` silenciosos. El programa simula la configuración de una aplicación, donde algunas opciones son obligatorias y otras tienen valores por defecto.

```

# Sintaxis moderna con símbolos
config = { host: "localhost", puerto: 3000 }

# Sintaxis hash rocket con strings (equivalente en espíritu)
config_vieja = { "host" => "localhost", "puerto" => 3000 }

puts config[:host]           # "localhost"
puts config.fetch(:puerto, 80) # 3000 (existe, usa el valor real)
puts config.fetch(:usuario, "admin") # "admin" (no existe, usa el defecto)

# Esto tronaría con KeyError, así que lo mostramos comentado:
# puts config.fetch(:password)

```

La lección: fetch con valor por defecto es tu red de seguridad para opciones configurables. El programa nunca truena por una clave faltante y siempre tiene un comportamiento sensato.

Ejemplo 3: carrito de compras con métodos de hash. Un carrito donde las claves son productos y los valores las cantidades. Usamos keys, values, merge y delete en un flujo realista: agregar productos, quitar uno, combinar con otro carrito y calcular el total de artículos con sum.

```

carrito = { manzana: 3, pan: 2, leche: 1 }

# Agregar más manzanas (modificar valor existente)
carrito[:manzana] += 2

# Quitar el pan
carrito.delete(:pan)

# Otro carrito para combinar
ofertas = { queso: 1, manzana: 1 }
# merge con bloque: si la clave se repite, suma las cantidades
total = carrito.merge(ofertas) { |producto, viejo, nuevo| viejo + nuevo }

puts "Productos: #{total.keys.inspect}"
puts "Cantidades: #{total.values.inspect}"
puts "Total de artículos: #{total.values.sum}"

```

El detalle elegante es merge con bloque: cuando una clave existe en ambos hashes, el bloque decide cómo combinar los valores (aquí los suma). Sin el bloque, el segundo hash simplemente aplastaría al primero. Este patrón aparece en reportes, inventarios y cualquier consolidación de datos.

Ejemplo 4: contador de palabras con each. Un clásico de entrevistas de trabajo: dado un texto, contar cuántas veces aparece cada palabra usando un hash. Muestra el patrón "hash como acumulador", quizá el uso más común de los hashes en algoritmos.

```
texto = "el perro corre y el gato duerme y el perro ladra"
conteo = {}

texto.split.each do |palabra|
  if conteo.key?(palabra)
    conteo[palabra] += 1
  else
    conteo[palabra] = 1
  end
end

conteo.each do |palabra, veces|
  puts "#{palabra}: #{veces}"
end

# el: 3, perro: 2, corre: 1, y: 2, gato: 1, duerme: 1, ladra: 1
```

El truco está en `key?`: si la palabra ya está en el hash, incrementamos; si no, la inicializamos en 1. (En Ruby avanzado esto se abrevia con `Hash.new(0)`, pero la versión explícita te enseña la mecánica.) Cada vez que necesites "contar cosas por categoría", este es el esqueleto.

Ejemplo 5: métodos con parámetros y retorno implícito. Tres métodos pequeños que muestran parámetros posicionales, el retorno implícito y cómo componer métodos (un método que usa a otro). Nada de `puts` dentro de los métodos que calculan: devuelven valores y el programa principal decide imprimirlos.

```

def area_rectangulo(base, altura)
  base * altura
end

def area_circulo(radio)
  3.1416 * radio * radio
end

def area_total_rectangulos(rectangulos)
  total = 0
  rectangulos.each do |r|
    total += area_rectangulo(r[:base], r[:altura])
  end
  total
end

puts area_rectangulo(5, 3)      # 15
puts area_circulo(2)           # 12.5664

rects = [{ base: 2, altura: 3 }, { base: 4, altura: 5 }]
puts area_total_rectangulos(rects) # 26

```

Observa cómo `area_total_rectangulos` reutiliza `area_rectangulo` en lugar de repetir la fórmula. Esa es la gracia de los métodos: escribes la lógica una vez y la compones. También nota que los rectángulos se representan como hashes dentro de un arreglo: estructuras anidadas simples y legibles.

Ejemplo 6: parámetros por defecto en un generador de reportes. Un método que imprime un reporte con formato; el título y el ancho tienen valores por defecto sensatos, pero se pueden personalizar. Muestra cómo los defaults hacen la llamada común ultracorta y la llamada especial expresiva.

```
def reporte(datos, titulo = "Reporte", ancho = 30)
  puts titulo.center(ancho, "-")
  datos.each do |clave, valor|
    puts "#{clave}: #{valor}"
  end
  puts "-" * ancho
end

ventas = { enero: 1200, febrero: 1500, marzo: 900 }

reporte(ventas)
reporte(ventas, "Ventas Q1", 40)
```

La primera llamada usa los dos defaults; la segunda personaliza todo. El método creció en capacidad sin romper a nadie: ese es el poder de los parámetros por defecto para evolucionar código existente.

Ejemplo 7: splat para una calculadora flexible. Un método promedio que acepta cualquier cantidad de números gracias a `*args`, más una demostración de desempaquetar un arreglo con `*` en la llamada. Incluye el caso borde de cero argumentos, manejado con un `return` temprano.

```
def promedio(*numeros)
  return 0 if numeros.empty?
  numeros.sum / numeros.length.to_f
end

puts promedio(10, 8, 9)           # 9.0
puts promedio(5, 5, 5, 5, 10)    # 6.0
puts promedio()                  # 0 (sin trueque)

calificaciones = [7, 8, 9, 10]
puts promedio(*calificaciones)    # 8.5 (desempaquetado)
```

Dos lecciones en uno: `*numeros` en la definición EMPAQUETA argumentos en un arreglo; `*calificaciones` en la llamada DESEMPAQUETA un arreglo en argumentos. Son operaciones espejo y dominarlas te da mucha flexibilidad.

Ejemplo 8: argumentos de palabra clave en un registro de usuarios. Un método `crear_usuario` con nombre y email obligatorios (keywords) y rol y activo opcionales con defaults. Las llamadas son autoexplicativas y el orden no importa. Incluye validación con `return` temprano.

```
def crear_usuario(nombre:, email:, rol: "lector", activo: true)
  return "Email inválido" unless email.include?("@")

  {
    nombre: nombre,
    email: email,
    rol: rol,
    activo: activo,
    creado: Time.now.strftime("%Y-%m-%d")
  }
end

u1 = crear_usuario(nombre: "Ana", email: "ana@mail.com")
u2 = crear_usuario(email: "jefe@mail.com", nombre: "Jefe", rol: "admin")

puts u1.inspect
puts u2.inspect
puts crear_usuario(nombre: "X", email: "sin-arroba")
```

Fíjate en tres cosas: el método devuelve un hash (los métodos pueden devolver cualquier cosa), la validación usa return temprano para despachar el caso malo al inicio, y las llamadas se leen como frases: crear_usuario(nombre: "Ana", email: ...). Cero ambigüedad de orden.

Ejemplo 9: convenciones ? y ! en una clase de texto. Un pequeño conjunto de métodos que demuestra predicados con ? y mutación con !. El programa muestra el antes y después de cada llamada para que veas exactamente qué cambia y qué no.

```

def titulo_valido?(titulo)
  !titulo.nil? && titulo.length >= 5
end

def gritar!(texto)
  texto.upcase!
  texto << "!!!"
end

def gritar(texto)
  texto.upcase + "!!!"
end

puts titulo_valido?("Hola")          # false (muy corto)
puts titulo_valido?("Hola mundo") # true

mensaje = "cuidado"
puts gritar(mensaje)  # "CUIDADO!!!" (mensaje sigue intacto)
puts mensaje          # "cuidado"

gritar!(mensaje)
puts mensaje          # "CUIDADO!!!" (mensaje mutó)

```

La moraleja visual: gritar devuelve algo nuevo y deja el original en paz; gritar! transforma el original. El ? de titulo_valido? anuncia "te respondo sí o no". Usa estas convenciones en tu propio código y quienes lo lean te lo van a agradecer.

Ejemplo 10: programa completo que integra todo. Un mini sistema de calificaciones: un hash de alumnos con arreglos de calificaciones, métodos con defaults y keywords para calcular promedios, splat para agregar calificaciones, y un reporte final. Es el examen de la unidad en forma de programa.

```

def promedio(*nums)
  return 0 if nums.empty?
  nums.sum / nums.length.to_f
end

def agregar_calificacion(alumnos, nombre:, calificacion:)
  alumnos[nombre] = [] unless alumnos.key?(nombre)
  alumnos[nombre] << calificacion
end

def reporte(alumnos, aprobatorio = 6.0)
  puts "--- Reporte de calificaciones ---"
  alumnos.each do |nombre, califs|
    prom = promedio(*califs)
    estado = prom >= aprobatorio ? "APROBADO" : "REPROBADO"
    puts "#{nombre}: #{prom.round(2)} (#{estado})"
  end
end

alumnos = { ana: [9, 8, 10], luis: [5, 6, 4] }
agregar_calificacion(alumnos, nombre: :ana, calificacion: 7)
agregar_calificacion(alumnos, nombre: :pedro, calificacion: 8)

reporte(alumnos)
reporte(alumnos, 8.0)

```

Este programa usa hashes (alumnos), each con dos parámetros, métodos con splat, keywords con defaults, retorno implícito, return temprano y el operador ternario. Si lo entiendes línea por línea sin ayuda, la unidad está dominada.

Práctica

Ejercicio 1: Crea un hash llamado libro con sintaxis de símbolos que tenga las claves titulo, autor y paginas con los valores "Cien años de soledad", "García Márquez" y 471. Luego imprime el autor usando acceso con símbolo.

Solución explicada:

```

libro = { titulo: "Cien años de soledad", autor: "García Márquez", paginas: 471 }
puts libro[:autor]

```

Usamos la sintaxis moderna { clave: valor } porque las claves son símbolos. Para leer, libro[:autor] con símbolo, no con string: recuerda que :autor y "autor" son claves distintas. Si hubieras escrito libro["autor"] obtendrías nil aunque el dato esté ahí.

Ejercicio 2: Dado el hash precios = { manzana: 10, pera: 12 }, agrega la clave uva con valor 18, cambia la pera a 15 y elimina la manzana. Imprime el hash final con inspect.

Solución explicada:

```
precios = { manzana: 10, pera: 12 }
precios[:uva] = 18      # agregar: la clave no existía
precios[:pera] = 15     # modificar: la clave ya existía
precios.delete(:manzana) # eliminar
puts precios.inspect    # {:pera=>15, :uva=>18}
```

La asignación con corchetes sirve para agregar y modificar sin distinción; delete quita el par completo. Nota que delete devuelve el valor eliminado (10), por si lo necesitas.

Ejercicio 3: Recorre el hash edades = { ana: 25, luis: 30, maria: 22 } con each e imprime "Ana tiene 25 años" (nombre capitalizado) para cada persona. Usa el método capitalize en la clave.

Solución explicada:

```
edades = { ana: 25, luis: 30, maria: 22 }
edades.each do |nombre, edad|
  puts "#{nombre.capitalize} tiene #{edad} años"
end
```

El bloque recibe |nombre, edad|: clave primero, valor después. nombre es un símbolo (:ana), y los símbolos también responden a capitalize, que devuelve "Ana". Si hubieras usado un solo parámetro en el bloque, recibirías arreglos [clave, valor] y tendrías que desarmarlos a mano.

Ejercicio 4: Combina los hashes a = { x: 1, y: 2 } y b = { y: 20, z: 3 } con merge en un hash nuevo, sin modificar a ni b. Imprime los tres hashes para comprobar que a y b quedaron intactos.

Solución explicada:

```
a = { x: 1, y: 2 }
b = { y: 20, z: 3 }
c = a.merge(b)
puts c.inspect # {:x=>1, :y=>20, :z=>3}
puts a.inspect # {:x=>1, :y=>2} (intacto)
puts b.inspect # {:y=>20, :z=>3} (intacto)
```

merge devuelve un hash NUEVO; en la clave repetida :y gana el valor de b (20). Si hubieras usado merge! (con admiración), el hash a sí se habría modificado. La versión sin ! es la segura por defecto.

Ejercicio 5: Define un método rectangulo(base, altura) que devuelva el área y el perímetro... no, mejor: que devuelva el área. Luego define cuadrado(lado) que reutilice rectangulo en lugar de repetir la fórmula. Prueba ambos.

Solución explicada:

```
def rectangulo(base, altura)
  base * altura
end

def cuadrado(lado)
  rectangulo(lado, lado)
end

puts rectangulo(4, 5) # 20
puts cuadrado(4)      # 16
```

El retorno es implícito: la última línea (base * altura) es el valor devuelto. cuadrado no repite la multiplicación: llama a rectangulo con el lado dos veces. Componer métodos así evita duplicar lógica; si la fórmula cambiara, solo la cambias en un lugar.

Ejercicio 6: Define un método despedir(nombre, mensaje = "Nos vemos") que imprima "Nos vemos, Ana." o el mensaje personalizado. Llámalo de las dos formas.

Solución explicada:


```
def despedir(nombre, mensaje = "Nos vemos")
  puts "#{mensaje}, #{nombre}."
end

despedir("Ana") # Nos vemos, Ana.
despedir("Ana", "Hasta luego") # Hasta luego, Ana.
```

El parámetro con default va al final. Cuando no pasas el segundo argumento, Ruby usa "Nos vemos". Este patrón es ideal para métodos con un comportamiento "normal" y variantes ocasionales.

Ejercicio 7: Define un método `maximo(*numeros)` que devuelva el número más grande recibido, o nil si no recibe ninguno. Pruébalo con varios argumentos y sin argumentos.

Solución explicada:

```
def maximo(*numeros)
  return nil if numeros.empty?
  numeros.max
end

puts maximo(3, 7, 2, 9) # 9
puts maximo(5).inspect  # 5
puts maximo().inspect   # nil
```

El splat `*numeros` empaqueta todo en un arreglo, así el método acepta cualquier cantidad. El `return` temprano maneja el caso borde (vacío) al inicio, y el flujo principal queda limpio con `numeros.max`. Sin el splat tendrías que pasar un arreglo explícito, menos cómodo.

Ejercicio 8: Define un método `ficha(nombre:, edad:, ciudad: "Desconocida")` que devuelva un string "Nombre (edad) - Ciudad". Llámalo omitiendo la ciudad y luego dándola, cambiando el orden de los argumentos en la segunda llamada.

Solución explicada:

```
def ficha(nombre:, edad:, ciudad: "Desconocida")
  "#{nombre} ({edad}) - #{ciudad}"
end

puts ficha(nombre: "Rosa", edad: 31)
# Rosa (31) - Desconocida
puts ficha(ciudad: "Puebla", edad: 31, nombre: "Rosa")
# Rosa (31) - Puebla
```

Las keywords obligatorias (nombre:, edad:) deben ir en la llamada o hay ArgumentError; ciudad: tiene default. El orden en la llamada no importa, lo que elimina la trampa de los posicionales. Si omites edad:, Ruby te dice exactamente qué faltó.

Ejercicio 9: Define un método par?(n) que devuelva true si n es par, y un método duplicar!(arreglo) que multiplique cada elemento por 2 modificando el arreglo original. Demuestra con puts antes y después que duplicar! sí muta.

Solución explicada:

```
def par?(n)
  n % 2 == 0
end

def duplicar!(arreglo)
  arreglo.map! { |x| x * 2 }
end

puts par?(4) # true
puts par?(7) # false

nums = [1, 2, 3]
puts nums.inspect # [1, 2, 3] (antes)
duplicar!(nums)
puts nums.inspect # [2, 4, 6] (después: mutó)
```

par? termina en ? porque responde una pregunta booleana. duplicar! termina en ! porque muta el objeto recibido (usa map!, la versión destructiva de map). Las convenciones avisan al lector qué esperar antes de leer el código.

Ejercicio 10: Escribe un método estadisticas(*numeros) que devuelva un hash con las claves cantidad, suma, promedio, maximo y minimo. Si no recibe números, el promedio debe ser 0. Pruébalo.

Solución explicada:

```
def estadisticas(*numeros)
  return { cantidad: 0, suma: 0, promedio: 0, maximo: nil, minimo: nil } if
  numeros.empty?

  {
    cantidad: numeros.length,
    suma: numeros.sum,
    promedio: numeros.sum / numeros.length.to_f,
    maximo: numeros.max,
    minimo: numeros.min
  }
end

puts estadisticas(4, 8, 6).inspect
# {:cantidad=>3, :suma=>18, :promedio=>6.0, :maximo=>8, :minimo=>4}
puts estadisticas().inspect
# {:cantidad=>0, :suma=>0, :promedio=>0, :maximo=>nil, :minimo=>nil}
```

Este ejercicio integra todo: splat para cantidad variable, return temprano para el caso vacío, retorno implícito de un hash multilinea y métodos de arreglo (sum, max, min). Devolver un hash con resultados nombrados es mucho más claro que devolver un arreglo [3, 18, 6.0, 8, 4] donde nadie sabe qué es cada posición.

Ejercicio 11: Dado el hash anidado tienda = { nombre: "Abarrotes Lupita", direccion: { ciudad: "Toluca" }, productos: [{ nombre: "Leche", precio: 25 }, { nombre: "Pan", precio: 10 }] }, imprime la ciudad con dig y calcula el precio total de los productos con sum y un bloque.

Solución explicada:

```
tienda = {
  nombre: "Abarrotes Lupita",
  direccion: { ciudad: "Toluca" },
  productos: [{ nombre: "Leche", precio: 25 }, { nombre: "Pan", precio: 10 }]
}

puts tienda.dig(:direccion, :ciudad) # "Toluca"
total = tienda[:productos].sum { |p| p[:precio] }
puts total # 35
```

dig recorre :direccion y luego :ciudad sin tronar aunque algo falte. tienda[:productos] es un arreglo de hashes, y sum con bloque extrae el :precio de cada uno y los suma. Es el patrón

"arreglo de hashes + enumerable" que domina el manejo de datos reales: catálogos, respuestas de API, resultados de base de datos.

Ejercicio 12: Define un método `buscar(hash, clave)` que devuelva el valor si la clave existe, o el string "no encontrado" si no existe, usando `key?` (no uses `fetch`). Pruébalo con una clave existente y una inexistente.

Solución explicada:

```
def buscar(hash, clave)
  return hash[clave] if hash.key?(clave)
  "no encontrado"
end

datos = { nombre: "Ana", edad: 30 }
puts buscar(datos, :nombre)      # "Ana"
puts buscar(datos, :telefono)    # "no encontrado"
```

El `return` temprano con `key?` evita el `nil` silencioso: en lugar de devolver `nil` (que podría confundirse con "el valor es `nil`"), devolvemos un mensaje explícito. La consigna prohibía `fetch` a propósito para que practiques `key?`; en código real, `fetch(:telefono, "no encontrado")` haría lo mismo en una línea. Conocer ambas formas te permite leer cualquier estilo.

Ejercicio 13: Define un método `normalizar(precios, factor)` que devuelva un hash NUEVO con cada precio multiplicado por `factor`, usando `transform_values`, sin modificar el hash original. Comprueba con `inspect` que el original quedó intacto.

Solución explicada:

```
def normalizar(precios, factor)
  precios.transform_values { |p| (p * factor).round(2) }
end

original = { manzana: 10, pera: 12.5 }
ajustado = normalizar(original, 1.16)

puts ajustado.inspect # {:manzana=>11.6, :pera=>14.5}
puts original.inspect # {:manzana=>10, :pera=>12.5} (intacto)
```

`transform_values` aplica el bloque a cada valor y devuelve un hash nuevo: el original no se toca. El `round(2)` evita decimales eternos al multiplicar por 1.16. Este es el patrón funcional para "ajustar todos los valores": nada de `each` manual construyendo un hash vacío; un método, un bloque, un hash nuevo. Y la comprobación del original intacto no es paranoia: es la

prueba de que entendiste la diferencia entre transformar y mutar, una distinción que en Ruby vale oro.

Resumen de la unidad

- Un hash es una colección de pares clave-valor: buscas por nombre (clave) en lugar de por posición, como en un diccionario o una agenda de contactos.
- Hay dos sintaxis: el hash rocket { "a" => 1 } (sirve para cualquier clave) y la sintaxis de símbolos { a: 1 } (atajo moderno cuando las claves son símbolos). :nombre y "nombre" son claves distintas: sé consistente.
- Lees con corchetes (hash[:clave]), escribes igual (hash[:clave] = valor crea o actualiza), y borras con delete. key? pregunta si existe una clave.
- each en hashes entrega clave y valor: hash.each do |clave, valor|. También existen each_key y each_value.
- Métodos clave: keys, values, merge (devuelve hash nuevo), merge! (modifica), fetch (lectura estricta con default opcional), length, empty?, slice, transform_values, dig para hashes anidados.
- Un método se define con def nombre ... end. Parámetros en la definición, argumentos en la llamada. Los posicionales dependen del orden: con 3 o más, prefiere palabras clave.
- Ruby tiene retorno implícito: la última línea del método es el valor devuelto. Usa return solo para salidas tempranas.
- Parámetros por defecto (def m(a, b = 1)) dan un valor cuando no pasas el argumento; van al final.
- El splat *args empaqueta argumentos variables en un arreglo; *arreglo en la llamada los desempaqueta.
- Los argumentos de palabra clave (def m(nombre:, edad: 0)) se llaman por nombre, no importa el orden, y los obligatorios lanzan error si faltan.
- Convenciones: ? para métodos que responden sí/no, ! para métodos que mutan o son la versión peligrosa/estricta.
- Buenas prácticas: un método una responsabilidad, nombres descriptivos en snake_case, métodos cortos, devolver valores en vez de imprimir, pocos parámetros, y nunca duplicar lógica: si se repite, es un método.

- Los hashes se anidan: un valor puede ser otro hash o un arreglo, formando estructuras como arreglos de hashes (el formato típico de APIs y bases de datos). Navega rutas profundas con dig para no tronar con nil intermedios, y evita pasar de 3 niveles: si la estructura crece más, es señal de que necesitas objetos.

Para cerrar esta unidad, quédate con una idea: los hashes te dan datos con significado y los métodos te dan comportamiento reutilizable. Cuando los combines, tus programas van a dejar de ser listas de instrucciones para convertirse en sistemas: datos organizados por nombre, transformados por métodos pequeños y honestos. La próxima vez que veas un arreglo de strings misteriosos donde cada posición "significa" algo, conviértelo en un arreglo de hashes. La próxima vez que copies un bloque de código por segunda vez, conviértelo en un método. Esos dos reflejos, practicados a diario, son los que separan al que "sabe sintaxis" del que "sabe programar". Y en la Unidad 6 vas a dar el salto final: empaquetar datos y comportamiento juntos en objetos, que es donde Ruby realmente brilla.

UNIDAD 6: Bloques y objetos

En la unidad anterior aprendiste a guardar datos con nombre (hashes) y a empaquetar comportamiento con nombre (métodos). Ahora viene la parte donde Ruby se vuelve Ruby de verdad: los bloques y los objetos. Si la Unidad 5 te dio herramientas de programador competente, esta unidad te da las herramientas de programador Ruby. Los bloques son la razón por la que el código Ruby se lee tan elegante, y los objetos son la razón por la que puedes construir programas grandes sin que se te caigan a pedazos.

Un bloque es un pedazo de código sin nombre que le pasas a un método para que lo ejecute cuando quiera, cuantas veces quiera. Ya los has usado sin darte cuenta: cada vez que escribiste `[1, 2, 3].each do |n| ... end`, ese `do...end` era un bloque. Es hora de entender qué son por dentro, cómo un método los recibe con `yield`, cuándo usar `do/end` y cuándo llaves, y cómo guardarlos en variables con `procs` y `lambdas`. Los bloques están en el corazón de Ruby: iteradores, configuración, manejo de recursos y los famosos DSLs (lenguajes específicos de dominio) como Rails y RSpec viven de ellos.

La segunda mitad de la unidad es la programación orientada a objetos: clases, objetos, constructores, variables de instancia y atributos. La idea central es sencilla pero poderosa: en lugar de tener datos por un lado (hashes sueltos) y funciones por otro (métodos sueltos), los unes en una sola cosa que tiene datos Y comportamiento. Un objeto `CuentaBancaria` guarda el saldo y sabe cómo depositar y retirar. Un objeto `Persona` guarda el nombre y sabe cómo presentarse. Esa unión hace que los programas grandes sean comprensibles: cada objeto cuida sus propios datos y ofrece sus propios servicios.

Vamos a construir esto con calma y con muchas analogías de la vida real, porque la POO es de esos temas que o se entienden con una buena analogía o no se entienden. Al final de la unidad vas a diseñar tus propias clases, con constructores, métodos y atributos, y vas a entender por qué Ruby es un lenguaje orientado a objetos "de verdad", donde hasta los números son objetos.

Explicación detallada

Qué es un bloque: código sin nombre que viaja

Un bloque es un fragmento de código que no tiene nombre propio y que se lo entregas a un método para que el método lo ejecute. Piensa en un bloque como una nota con instrucciones que le das a alguien: "cuando termines de cocinar, sigue estas instrucciones". Tú escribes las instrucciones, pero es el otro quien decide cuándo ejecutarlas.

Ya usaste bloques desde las primeras unidades, aunque quizá no los llamabas así:

```
[1, 2, 3].each do |n|  
  puts n * 2  
end
```

El bloque es todo lo que está entre `do` y `end`. Se lo pasas al método `each`, y `each` lo ejecuta una vez por cada elemento, entregándole el elemento en la variable `n`. Tú no controlas el ciclo: `each` decide cuándo correr tu código. Esa inversión de control es la esencia de los bloques: tú describes QUÉ hacer con cada elemento, y el método se encarga del CÓMO y el CUÁNDO.

Los bloques tienen dos sintaxis: `do/end` para bloques multilínea y llaves `{ }` para bloques de una línea (aunque técnicamente puedes usar cualquiera en cualquier caso):

```
# Multilínea con do/end  
[1, 2, 3].each do |n|  
  puts n * 2  
end  
  
# Una línea con llaves  
[1, 2, 3].each { |n| puts n * 2 }
```

Ambos hacen exactamente lo mismo. La elección entre uno y otro es convención de estilo, que veremos en detalle más adelante.

Lo revolucionario de los bloques es que convierten a los métodos en plantillas de comportamiento. Sin bloques, `each` tendría que saber qué quieres hacer con cada elemento, lo cual es imposible porque cada programa quiere algo distinto. Con bloques, `each` solo sabe recorrer, y TÚ le inyectas la acción. Es como una licuadora: la licuadora sabe girar (el método), tú decides qué frutas echarle (el bloque).

Los bloques aparecen en todas partes en Ruby: `each`, `map`, `select`, `times`, `File.open`, `sort`, y miles de métodos de gemas. Cuando veas un método seguido de `do/end` o de llaves, ya sabes: está recibiendo un bloque. Y en la siguiente sección vas a aprender a escribir tus propios métodos que reciben bloques, que es donde la magia se vuelve herramienta.

yield: el botón que ejecuta tu bloque

Si los bloques son notas con instrucciones, yield es el botón que el método aprieta para ejecutarlas. Cuando defines un método, no declaras el bloque como parámetro: simplemente llamas a yield dentro del método, y Ruby ejecuta el bloque que le pasaron en la llamada.

```
def cocinar
  puts "Prendiendo la estufa..."
  yield # Aquí se ejecuta el bloque
  puts "Apagando la estufa."
end

cocinar do
  puts "Friendo los huevos"
end

# Prendiendo la estufa...
# Friendo los huevos
# Apagando la estufa.
```

Fíjate en el flujo: cocinar empieza, hace su parte, aprieta yield (corre tu bloque), y luego continúa. El método envuelve tu código con comportamiento previo y posterior. Este patrón se llama "sándwich de código" y es increíblemente útil: abrir un archivo, ejecutar tu bloque, cerrar el archivo; iniciar una transacción, ejecutar tu bloque, confirmar o revertir; medir el tiempo, ejecutar tu bloque, reportar cuánto tardó.

yield también puede pasarle argumentos al bloque. Los valores que le des a yield llegan a los parámetros del bloque entre plecas:

```
def repetir(vecas)
  i = 1
  while i <= vecas
    yield(i) # le pasa el número actual al bloque
    i += 1
  end
end

repetir(3) do |n|
  puts "Vuelta número #{n}"
end

# Vuelta número 1
# Vuelta número 2
# Vuelta número 3
```

Acabamos de reinventar times: repetir(3) hace lo mismo que 3.times. El método controla el ciclo (el while), y el bloque decide qué hacer en cada vuelta con el valor que recibe. Separa responsabilidades de forma hermosa.

También puedes preguntar si te pasaron un bloque con block_given?, para comportarte distinto en cada caso:

```
def saludar
  if block_given?
    yield
  else
    puts "Hola (sin bloque)"
  end
end

saludar { puts "Hola con bloque" } # Hola con bloque
saludar                               # Hola (sin bloque)
```

Esto hace tus métodos flexibles: funcionan con o sin bloque. Muchas gemas usan este patrón para métodos que pueden usarse de forma simple o avanzada.

Y puedes llamar a yield varias veces, o ninguna, o dentro de condicionales. El método tiene control total sobre CUÁNDO y CUÁNTAS veces se ejecuta tu bloque. Por ejemplo, un método que ejecuta el bloque solo si una condición se cumple:

```
def solo_si_es_par(n)
  yield(n) if n.even?
end

solo_si_es_par(4) { |x| puts "#{x} es par" } # se ejecuta
solo_si_es_par(5) { |x| puts "#{x} es par" } # no se ejecuta, silencio total
```

Error común de principiantes: llamar a yield en un método sin pasarle bloque. Eso lanza LocalJumpError. Si tu método usa yield, o documenta que el bloque es obligatorio, o protege con block_given?. No hay término medio silencioso: sin bloque, yield truena.

Otro error: pensar que el valor de yield se pierde. No se pierde: yield DEVUELVE lo que devuelva el bloque (su última expresión), y el método puede usar ese valor:

```
def doble_bloque
  resultado = yield
  puts "El bloque devolvió: #{resultado}"
end

doble_bloque { 21 * 2 } # El bloque devolvió: 42
```

Esto permite patrones avanzados donde el método transforma o decide con base en lo que el bloque produce, como `select` (que guarda el elemento solo si el bloque devuelve algo verdadero).

do/end vs llaves: la convención que evita dolores de cabeza

Técnicamente, `do/end` y `{ }` son intercambiables: ambos crean bloques. Pero la comunidad Ruby tiene una convención clara y conviene seguirla porque el código ajeno la sigue: usa llaves `{ }` para bloques de UNA línea (enfocados en el valor que devuelven) y `do/end` para bloques MULTILÍNEA (enfocados en los efectos que producen).

```
# Una línea, me importa el resultado: llaves
dobles = [1, 2, 3].map { |n| n * 2 }

# Multilínea, me importan los efectos: do/end
[1, 2, 3].each do |n|
  puts "Procesando #{n}..."
  guardar_en_base(n)
end
```

La lógica detrás: `map` transforma y devuelve algo nuevo (enfoque funcional, una línea, llaves); `each` ejecuta acciones con efectos secundarios (imprimir, guardar), normalmente en varias líneas (`do/end`). No es una regla del lenguaje, es una guía de legibilidad: cuando ves llaves esperas una transformación corta; cuando ves `do/end` esperas un procedimiento.

Pero hay una diferencia técnica real que SÍ importa, y es la precedencia: las llaves se pegan más fuerte al método que las precede que `do/end`. Mira este caso clásico que confunde a todos los principiantes:

```
# Con llaves: el bloque se pega a select
puts [1, 2, 3, 4].select { |n| n.even? }.inspect
# [2, 4] (correcto)

# Con do/end: el bloque se pega a puts, ¡no a select!
puts [1, 2, 3, 4].select do |n|
  n.even?
end.inspect
# Esto imprime el arreglo COMPLETO, porque el bloque se lo quedó puts
```

En el segundo caso, `do/end` se asocia con `puts` (el método más externo), no con `select`. El `select` se ejecuta SIN bloque (devuelve un enumerador raro) y el bloque se lo queda `puts`, que lo ignora. El resultado es confuso y es una fuente clásica de bugs. Regla práctica para no sufrir: si el bloque va en una cadena de métodos o en la misma línea que otro método, usa llaves. Si el bloque está en su propia estructura multilínea, usa `do/end`.

```
# Código malo: do/end pegado a puts por accidente
puts [1, 2, 3].map do |n|
  n * 10
end

# Código bueno: llaves para bloques en línea con otros métodos
puts([1, 2, 3].map { |n| n * 10 }.inspect)

# Código bueno: do/end en su propio bloque separado
resultado = [1, 2, 3].map do |n|
  n * 10
end
puts resultado.inspect
```

Cuando tengas duda, pon paréntesis explícitos o asigna a una variable primero. La claridad le gana siempre a la elegancia críptica.

Parámetros de bloque: las `[plecas]` que reciben valores

Los parámetros de bloque viven entre plecas `| |` al inicio del bloque y reciben los valores que el método les pasa con `yield`. Ya los viste en `each` (`|n|`), pero hay matices que vale la pena dominar.

Primero: puedes recibir varios parámetros si el método hace `yield` con varios valores. `each` en hashes es el ejemplo clásico:

```
{ nombre: "Ana", edad: 30 }.each do |clave, valor|
  puts "#{clave} => #{valor}"
end
```

`each_with_index` es otro: te entrega el elemento y su índice.

```
["a", "b", "c"].each_with_index do |letra, i|
  puts "#{i}: #{letra}"
end
# 0: a
# 1: b
# 2: c
```

Segundo: si el método pasa más valores de los que el bloque declara, los extra se ignoran en silencio. Si el bloque declara más parámetros de los que recibe, los faltantes son nil. Ruby es tolerante aquí, a diferencia de los métodos normales que exigen el número exacto:

```
def tres_valores
  yield(1, 2, 3)
end

tres_valores { |a| puts a } # 1 (ignora 2 y 3)
tres_valores { |a, b, c, d| puts d.inspect } # nil (d no recibió nada)
```

Tercero: los parámetros de bloque son locales al bloque. No pisan variables de afuera aunque se llamen igual... bueno, casi. En Ruby moderno (1.9+), los parámetros del bloque SIEMPRE son locales nuevos, incluso si afuera existe una variable con el mismo nombre. Pero las variables que CREAS dentro del bloque (sin ser parámetros) sí pueden pisar las de afuera si se llaman igual. Mira la diferencia:

```
x = 100
[1, 2, 3].each do |x| # parámetro: es una x NUEVA, local al bloque
  puts x # 1, 2, 3 (la x de afuera sigue en 100)
end
puts x # 100 (intacta)

[1, 2, 3].each do |n|
  x = n * 10 # NO es parámetro: pisa la x de afuera
end
puts x # 30 (la de afuera cambió)
```

Para evitar el segundo caso (pisar sin querer), puedes declarar variables locales del bloque después de un punto y coma en las plecas: `|n; y|`. Esas variables son locales al bloque aunque exista una igual afuera. Se usa poco, pero existe por si lo necesitas.

Cuarto: puedes dar valores por defecto a parámetros de bloque, igual que en métodos, aunque es raro verlo. Y puedes usar `splat` en parámetros de bloque para recoger valores extra en un arreglo. Lo importante es que entiendas el mecanismo: `yield` envía, las plecas reciben, y los nombres los eliges tú para que el código se lea bien.

Procs: bloques embotellados que puedes guardar

Un bloque normal vive pegado a la llamada de un método: nace y muere ahí. Un proc es un bloque que embotellaste: lo guardaste en una variable para usarlo después, pasarlo a otros métodos o ejecutarlo cuantas veces quieras. Es la diferencia entre una nota que le das a alguien en la mano (bloque) y una nota que guardas en tu bolsillo para usarla cuando quieras (proc).

Se crea con `Proc.new` y se ejecuta con `call`:

```
saludar = Proc.new { |nombre| puts "Hola, #{nombre}" }

saludar.call("Ana")      # Hola, Ana
saludar.call("Luis")     # Hola, Luis
```

También puedes crearlos con la sintaxis `proc` (más corta) o con el operador `lambda`, pero de `lambdas` hablamos en la siguiente sección. Por ahora: `Proc.new { ... }` guarda el bloque en una variable, y `.call` lo ejecuta.

¿Para qué sirve guardar un bloque? Para reutilizarlo en varios lugares sin repetirlo:

```
# Sin proc: el bloque se repite en cada llamada
puts [1, 2, 3].map { |n| n * 2 }.inspect
puts [4, 5, 6].map { |n| n * 2 }.inspect

# Con proc: lo defines una vez y lo reutilizas
doble = Proc.new { |n| n * 2 }
puts [1, 2, 3].map(&doble).inspect
puts [4, 5, 6].map(&doble).inspect
```

Fíjate en el & delante de doble: ese ampersand convierte el proc en bloque para que map lo acepte. El & es el puente entre los dos mundos: &proc convierte un proc en bloque, y en la definición de un método, &bloque convierte un bloque en proc. Es el mismo símbolo haciendo el viaje de ida y vuelta:

```
def ejecutar(&bloque) # el bloque de la llamada se guarda como proc
  puts "Antes"
  bloque.call        # lo ejecutamos cuando queremos
  puts "Después"
end

ejecutar { puts "Soy el bloque" }
# Antes
# Soy el bloque
# Después
```

Con &bloque en la definición, el método recibe el bloque como un objeto proc normal: puedes guardarlo, pasarlo a otro método o ejecutarlo varias veces. Sin el &, solo puedes usar yield una vez por llamada... bueno, puedes hacer yield varias veces, pero no puedes guardar el bloque para después. El parámetro &bloque te da el control total.

Otro uso poderoso: pasar el mismo comportamiento a métodos distintos o almacenarlo en estructuras de datos. Imagina un menú de operaciones:

```
operaciones = {
  sumar: Proc.new { |a, b| a + b },
  restar: Proc.new { |a, b| a - b },
  multiplicar: Proc.new { |a, b| a * b }
}

puts operaciones[:sumar].call(5, 3)      # 8
puts operaciones[:multiplicar].call(5, 3) # 15
```

Un hash de procs es básicamente una tabla de comportamientos elegibles por nombre. Este patrón aparece en máquinas de estado, calculadoras, manejadores de comandos y menús de programas.

Lambdas: los primos estrictos de los procs

Una lambda es casi lo mismo que un proc: un bloque guardado en una variable que ejecutas con call. Se crea con la palabra lambda (o con la sintaxis flecha ->, que es su forma moderna y corta):

```
doble = lambda { |n| n * 2 }  
triple = ->(n) { n * 3 } # sintaxis flecha: ->(parámetros) { código }  
  
puts doble.call(5) # 10  
puts triple.call(5) # 15
```

Entonces, ¿para qué existen dos cosas casi iguales? Porque tienen DOS diferencias importantes de comportamiento, y esas diferencias hacen que cada uno sirva para cosas distintas. Las vemos a fondo en la siguiente sección, pero el resumen es: las lambdas son estrictas con los argumentos (como los métodos) y su return solo sale de la lambda; los procs son tolerantes con los argumentos y su return intenta salir del método que los contiene.

¿Cuándo usar lambda? Cuando quieres algo que se comporte como un método anónimo: validación estricta de argumentos y retorno local. Las lambdas son ideales para callbacks y funciones pequeñas que pasas por ahí:

```
# Lambda como validador reutilizable  
mayor_de_edad = ->(edad) { edad >= 18 }  
  
puts mayor_de_edad.call(20) # true  
puts mayor_de_edad.call(15) # false
```

La sintaxis de flecha `->(edad) { ... }` se lee casi como una definición de método miniatura, y esa es la idea: una lambda ES un método sin nombre. De hecho, muchos rubyistas piensan en lambdas como "métodos que puedes guardar en variables", mientras que los procs son "bloques que puedes guardar en variables". La distinción es sutil pero útil para elegir.

Diferencias entre procs y lambdas: las dos que sí importan

Vamos al grano con las dos diferencias, porque son pregunta clásica de entrevista y fuente de bugs si las ignoras.

Diferencia 1: verificación de argumentos. La lambda es estricta como un método: si le pasas más o menos argumentos de los que declara, truena con `ArgumentError`. El proc es tolerante como un bloque: los argumentos de más se ignoran y los de menos se vuelven `nil`.


```

# Proc: tolerante
p = Proc.new { |a, b| puts "a=#{a.inspect}, b=#{b.inspect}" }
p.call(1)          # a=1, b=nil (no truena)
p.call(1, 2, 3)    # a=1, b=2 (ignora el 3)

# Lambda: estricta
l = lambda { |a, b| puts "a=#{a.inspect}, b=#{b.inspect}" }
l.call(1, 2)       # a=1, b=2 (bien)
# l.call(1)        # ArgumentError: wrong number of arguments
# l.call(1, 2, 3)  # ArgumentError: wrong number of arguments

```

Diferencia 2: comportamiento de return. En una lambda, return sale SOLO de la lambda, como en un método normal. En un proc, return intenta salir del MÉTODO que contiene al proc (el contexto donde se definió), lo cual puede tronar si ese método ya terminó.

```

def prueba_proc
  p = Proc.new { return "salí del método" }
  p.call
  "nunca llego aquí"
end

def prueba_lambda
  l = lambda { return "salí solo de la lambda" }
  resultado = l.call
  "el método sigue: #{resultado}"
end

puts prueba_proc    # "salí del método" (el return escapó del método)
puts prueba_lambda  # "el método sigue: salí solo de la lambda"

```

El return del proc es como un botón de eyección: no sale del proc, sale de todo el método donde el proc fue definido. Esto es herencia de los bloques (en un bloque normal, return también sale del método que lo contiene). La lambda, en cambio, se comporta como un método independiente: su return es local.

Esta diferencia hace que los procs sean peligrosos si los guardas y los ejecutas mucho después de que el método original terminó: el return intentaría salir de un método que ya no está en ejecución y Ruby lanza LocalJumpError. Las lambdas no tienen ese problema.

Tabla mental para decidir:

```
# Usa lambda cuando...
# - quieres validación estricta de argumentos
# - el código se comporta como un método miniatura
# - vas a guardar el callable por mucho tiempo
validador = ->(x) { x > 0 }

# Usa proc (o bloque) cuando...
# - quieres tolerancia con los argumentos
# - es un iterador o callback simple
# - necesitas el return que escapa (caso raro y avanzado)
iterador = Proc.new { |x| puts x }
```

En la práctica moderna de Ruby, las lambdas con sintaxis `->` son las más usadas para callables guardados, y los bloques normales (`do/end`, `{}`) para iteración. Los `Proc.new` explícitos aparecen más en metaprogramación y código avanzado. Conocer los tres te permite leer cualquier código Ruby sin miedo.

Qué es la programación orientada a objetos: el cambio de chip

Hasta ahora has programado de forma procedural: datos por aquí (variables, arreglos, hashes), instrucciones por allá (métodos sueltos que operan sobre esos datos). Funciona, pero tiene un problema de escala: cuando el programa crece, los datos y el código que los manipula se separan cada vez más, y cualquier parte del programa puede tocar cualquier dato. Es como una cocina donde todos los cocineros meten mano a todas las ollas: tarde o temprano alguien echa sal al postre.

La programación orientada a objetos (POO) propone otra organización: junta los datos y el comportamiento en unidades llamadas OBJETOS. Cada objeto cuida sus propios datos (nadie más los toca directamente) y ofrece métodos para interactuar con él. La cocina se organiza: cada cocinero tiene su estación, sus ingredientes y sus recetas; si quieres algo de otra estación, lo pides por la ventanilla (los métodos), no metes la mano en su olla.

Los cuatro pilares de la POO (los verás nombrados en todas partes):

1. Encapsulamiento: el objeto esconde sus datos internos y solo los expone a través de métodos. Tú no necesitas saber CÓMO una `CuentaBancaria` guarda el saldo; solo necesitas depositar y retirar. Si mañana cambia la implementación interna, tu código no se rompe.
2. Abstracción: te quedas con lo esencial e ignoras los detalles. Cuando manejas un coche usas volante y pedales, no piensas en la inyección de combustible. Un buen objeto ofrece una interfaz simple que oculta la complejidad interna.

3. Herencia: puedes crear clases nuevas basadas en clases existentes, heredando su comportamiento y agregando o modificando lo necesario. Un Perro hereda de Animal: ya sabe respirar y comer, solo agrega ladrar. (La herencia a fondo es tema avanzado; aquí la mencionamos para que el mapa esté completo.)
4. Polimorfismo: objetos distintos responden al mismo mensaje cada uno a su manera. Le pides a un Pato que haga cuac y a un Perro que haga cuac... bueno, mal ejemplo. Le pides a una impresora que imprima y a un archivo que "imprima" (guarde): el mismo mensaje, comportamientos distintos según el objeto.

¿Por qué Ruby es especial para POO? Porque en Ruby TODO es un objeto: los números, los strings, los arreglos, los hashes, hasta las clases mismas. No hay "tipos primitivos" de segunda clase como en otros lenguajes. Cuando escribes `5.times` o `"hola".upcase`, le estás enviando mensajes a objetos. Ya llevas toda la unidad (y el curso) haciendo POO sin saberlo; ahora vas a aprender a crear tus propios objetos.

La analogía que lo amarra todo: una CLASE es el molde para galletas, un OBJETO es cada galleta que sale del molde. El molde define la forma (qué datos tiene una galleta: tamaño, figura; qué puede hacer: hornearse, decorarse). Cada galleta es independiente: puedes decorar una sin afectar a las demás. En código: defines la clase `Persona` una vez, y creas tantos objetos `persona` como necesites (ana, luis, pedro), cada uno con sus propios datos pero compartiendo el mismo comportamiento.

Clases y objetos: el molde y las galletas

Vamos a lo concreto. Una clase se define con `class` y `end`, y su nombre empieza con mayúscula (CamelCase por convención: `Persona`, `CuentaBancaria`, `CarroDeportivo`). Un objeto se crea con el método `new`:

```
class Perro
end

firulais = Perro.new
solovino = Perro.new

puts firulais.inspect # #<Perro:0x00007f...>
puts solovino.inspect # #<Perro:0x00007f...> (otro objeto, distinto id)
```

Perro.new crea un objeto nuevo de la clase Perro. firulais y solovino son dos objetos DISTINTOS: cada uno tiene su propia identidad (fíjate en el código hexadecimal distinto en el inspect). Por ahora no hacen nada interesante porque la clase está vacía, pero ya son objetos independientes.

Los objetos cobran vida con los MÉTODOS DE INSTANCIA: métodos definidos dentro de la clase que cada objeto puede usar. Se definen con def igual que siempre, pero viven dentro del class/end:

```
class Perro
  def ladrar
    puts "¡Guau guau!"
  end

  def presentarse
    puts "Soy un perro y ladro así:"
    ladrar # un método puede llamar a otro del mismo objeto
  end
end

firulais = Perro.new
firulais.ladrar      # ¡Guau guau!
firulais.presentarse
# Soy un perro y ladro así:
# ¡Guau guau!
```

Fíjate en la sintaxis de llamada: objeto.metodo. El punto significa "envíale este mensaje a este objeto". firulais.ladrar le pide a firulais que ladre. solovino.ladrar le pediría a solovino. Mismo método, distintos objetos, cada uno actuando por su cuenta.

Pero un perro que solo ladra es aburrido: los objetos interesantes tienen DATOS propios. Cada perro debería tener su nombre, su edad, su raza. Esos datos viven en VARIABLES DE INSTANCIA, que empiezan con @ y pertenecen a un objeto específico (no se comparten entre objetos). Para darles valores iniciales usamos el método especial initialize, que Ruby llama automáticamente cuando haces new. Todo esto lo desmenuzamos en las siguientes secciones; por ahora quédate con el mapa:

```

class Perro
  def initialize(nombre)
    @nombre = nombre # variable de instancia: cada perro tiene la suya
  end

  def ladrar
    puts "#{@nombre} dice: ¡Guau guau!"
  end
end

firulais = Perro.new("Firulais")
solovino = Perro.new("Solovino")
firulais.ladrar # Firulais dice: ¡Guau guau!
solovino.ladrar # Solovino dice: ¡Guau guau!

```

El mismo método ladrar produce resultados distintos porque cada objeto tiene su propio @nombre. Esa es la magia resumida: comportamiento compartido (el método, definido una vez en la clase) + datos privados (las variables @, una copia por objeto).

Error común de principiantes: confundir la clase con el objeto. La clase es el PLANO, el objeto es la CASA construida con ese plano. No vives en el plano; vives en la casa. En código: no llamas Perro.ladrar (la clase no ladra), llamas firulais.ladrar (el objeto sí). Cuando veas "undefined method X for Class", casi siempre es que llamaste un método de instancia sobre la clase en lugar de sobre un objeto creado con new.

Otro error: crear el objeto pero olvidar guardarlo en una variable. Perro.new sin asignarlo crea un perro fantasma que nadie puede usar y Ruby lo desecha. Siempre: firulais = Perro.new.

initialize: el constructor que prepara cada objeto

Cuando creas un objeto con new, casi siempre quieres que nazca ya configurado: un Perro con nombre, una CuentaBancaria con saldo inicial, un Alumno con matrícula. El método especial initialize es el constructor: Ruby lo llama automáticamente justo después de crear el objeto, pasándole los argumentos que le diste a new.

```
class Perro
  def initialize(nombre, edad)
    @nombre = nombre
    @edad = edad
    puts "Nació #{@nombre}"
  end
end

firulais = Perro.new("Firulais", 3)
# Nació Firulais
```

El flujo es: `Perro.new("Firulais", 3)` crea el objeto vacío y LUEGO llama a `initialize("Firulais", 3)`. Los argumentos de `new` viajan directo a `initialize`. Por eso el número de argumentos debe cuadrar: si `initialize` pide dos y haces `new` con uno, hay `ArgumentError`.

La analogía: `initialize` es como el acta de nacimiento. Cuando nace un bebé (`new`), se registra su nombre y fecha (`initialize`). Nadie nace sin registro, y ningún objeto bien diseñado nace sin sus datos esenciales. Si tu objeto necesita ciertos datos para funcionar, pídelos en `initialize` y hazlos obligatorios: así es imposible crear un objeto "incompleto" que falle después.

Puedes usar valores por defecto y palabras clave en `initialize` igual que en cualquier método:

```
class CuentaBancaria
  def initialize(titular:, saldo: 0)
    @titular = titular
    @saldo = saldo
  end
end

c1 = CuentaBancaria.new(titular: "Ana") # saldo 0 por defecto
c2 = CuentaBancaria.new(titular: "Luis", saldo: 500) # saldo 500
```

Esto combina lo mejor de dos mundos: datos obligatorios (`titular:`) que garantizan objetos válidos, y opcionales con defaults (`saldo: 0`) que hacen la creación cómoda. Es el patrón estándar en Ruby moderno.

Un detalle importante: `initialize` es PRIVADO por defecto en el sentido de que no lo llamas directamente sobre un objeto (no escribes `firulais.initialize`). Solo Ruby lo invoca, una vez, al nacer el objeto. Tú solo decides QUÉ argumentos pasan por `new`.

Error común: olvidar que `initialize` existe y tratar de configurar el objeto después de crearlo con métodos sueltos. Funciona, pero deja una ventana donde el objeto existe a medias:

```
# Código malo: objeto incompleto entre new y configurar
p = Perro.new
# ... aquí el perro no tiene nombre, cualquier método que use @nombre falla ...
p.poner_nombre("Firulais")

# Código bueno: nace completo
p = Perro.new("Firulais")
```

Otro error: hacer trabajo pesado en initialize (conexiones a red, lecturas de archivos grandes). El constructor debe ser rápido y predecible: asignar datos y validar. Lo pesado va en métodos explícitos que el usuario llama cuando está listo.

Variables de instancia: el @ que hace cada objeto único

Las variables de instancia empiezan con @ y son el "cerebro privado" de cada objeto: cada objeto tiene su propia copia, invisible e inaccesible desde afuera. @nombre de firulais y @nombre de solovino son dos variables distintas que casualmente se llaman igual.

```
class Perro
  def initialize(nombre)
    @nombre = nombre
  end

  def ladrar
    puts "#{@nombre}: ¡Guau!"
  end

  def cambiar_nombre(nuevo)
    @nombre = nuevo
  end
end

a = Perro.new("Firulais")
b = Perro.new("Solovino")
a.cambiar_nombre("Max")
a.ladrar # Max: ¡Guau!
b.ladrar # Solovino: ¡Guau! (no se afectó)
```

Tres reglas de oro sobre las variables de instancia:

Regla 1: viven mientras viva el objeto. A diferencia de las variables locales de un método (que mueren cuando el método termina), las `@` persisten entre llamadas a métodos. Por eso ladrar puede usar `@nombre` aunque se asignó en `initialize`: la variable sobrevivió.

Regla 2: son privadas por defecto. Desde FUERA del objeto no puedes leer ni escribir `@nombre` directamente. Esto NO funciona:

```
f = Perro.new("Firulais")
puts f.@nombre # SyntaxError: syntax invalid
```

Desde fuera solo puedes interactuar a través de métodos. Esto es el encapsulamiento en acción: el objeto decide qué muestra y qué esconde. Si quieres exponer un dato, defines un método para ello (o usas `attr_reader`, que veremos enseguida).

Regla 3: si lees una `@` que nunca se asignó, vale `nil` (no truena). Esto es cómodo pero traicionero: un typo como `@nobre` en lugar de `@nombre` no da error, solo te da `nil` y un bug misterioso. Cuando un método te devuelva `nil` inesperado, revisa que el nombre de la variable de instancia esté bien escrito en TODOS los métodos.

Compara con los otros tipos de variables para no confundirlos:

```
$global = 1    # variable global: visible en TODO el programa (evítala)
@@clase = 2    # variable de clase: compartida entre la clase y sus objetos (úsala poco)
@instancia = 3 # variable de instancia: privada de cada objeto (tu pan de cada día)
local = 4      # variable local: vive solo en su método o bloque
```

En tu día a día usarás `@` y locales casi exclusivamente. Las globales (`$`) son mala práctica (cualquiera las puede cambiar desde cualquier lugar) y las de clase (`@@`) tienen comportamientos sorpresivos con herencia. Si sientes la tentación de usar una global, casi seguro lo que necesitas es una variable de instancia bien encapsulada.

Métodos de instancia: lo que cada objeto sabe hacer

Ya definiste métodos de instancia en los ejemplos (ladrar, presentarse): son los `def` dentro del `class/end`, y cada objeto los puede llamar con `objeto.metodo`. Pero vamos a profundizar en cómo trabajan con las variables de instancia y entre sí.

Dentro de un método de instancia, `self` es el objeto actual (el que recibió el mensaje). Casi nunca necesitas escribirlo explícito, pero entenderlo aclara mucho: cuando `firulais.ladrar` se ejecuta, dentro de `ladrar` `self` es `firulais`, y `@nombre` es el `@nombre` DE FIRULAI.


```

class Perro
  def initialize(nombre)
    @nombre = nombre
    @energia = 100
  end

  def jugar(minutos)
    @energia -= minutos * 2
    @energia = 0 if @energia < 0
    puts "#{@nombre} jugó #{minutos} minutos. Energía: #{@energia}"
  end

  def dormir(horas)
    @energia += horas * 10
    @energia = 100 if @energia > 100
    puts "#{@nombre} durmió #{horas} horas. Energía: #{@energia}"
  end

  def estado
    "#{@nombre} tiene #{@energia} de energía"
  end
end

f = Perro.new("Firulais")
f.jugar(10)  # Firulais jugó 10 minutos. Energía: 80
f.dormir(3)  # Firulais durmió 3 horas. Energía: 100
puts f.estado # Firulais tiene 100 de energía

```

Observa el diseño: `@energia` nunca se toca desde fuera. Los métodos `jugar` y `dormir` son la "ventanilla" que modifica la energía con reglas (no baja de 0, no sube de 100). Si alguien pudiera hacer `f.@energia = 9999` directamente, las reglas no servirían de nada. El encapsulamiento protege la coherencia interna del objeto.

Los métodos de instancia también pueden llamarse entre sí sin receptor explícito: dentro de `presentarse` escribimos `ladrar a secas`, y Ruby lo envía a `self` automáticamente. Esto permite componer comportamientos: métodos pequeños que se combinan en métodos más grandes, igual que en la unidad anterior pero ahora con acceso a los datos del objeto.

Un patrón fundamental: métodos que VALIDAN antes de actuar. Un objeto serio no deja que lo pongan en un estado inválido:

```

class CuentaBancaria
  def initialize(titular, saldo = 0)
    @titular = titular
    @saldo = saldo
  end

  def depositar(cantidad)
    return "Cantidad inválida" if cantidad <= 0
    @saldo += cantidad
    "Depósito exitoso. Saldo: #{@saldo}"
  end

  def retirar(cantidad)
    return "Cantidad inválida" if cantidad <= 0
    return "Fondos insuficientes" if cantidad > @saldo
    @saldo -= cantidad
    "Retiro exitoso. Saldo: #{@saldo}"
  end

  def saldo
    @saldo
  end
end

```

retirar no permite dejar la cuenta en negativo: la regla vive DENTRO del objeto, no en el código que lo usa. Quien use la cuenta no necesita acordarse de validar; el objeto se cuida solo. Ese es el poder del encapsulamiento: las reglas del negocio viven junto a los datos, no regadas por todo el programa.

attr_reader, attr_writer, attr_accessor: ventanillas automáticas

Recuerda: desde fuera no puedes tocar @nombre directamente. Si quieres permitir lectura, defines un método que la devuelva; si quieres permitir escritura, defines un método que la asigne:

```

class Perro
  def initialize(nombre)
    @nombre = nombre
  end

  def nombre      # getter: permite LEER @nombre
    @nombre
  end

  def nombre=(nuevo) # setter: permite ESCRIBIR @nombre
    @nombre = nuevo
  end
end

f = Perro.new("Firulais")
puts f.nombre      # "Firulais" (lectura)
f.nombre = "Max"   # escritura (fíjate: se ve como asignación normal)
puts f.nombre      # "Max"

```

Fíjate en la magia sintáctica: `f.nombre = "Max"` PARECE una asignación a variable, pero en realidad llama al método `nombre=`. Ruby te da sintaxis bonita para setters: `objeto.atributo = valor`.

Escribir getters y setters a mano para cada variable es tedioso y repetitivo, así que Ruby trae tres atajos que generan estos métodos automáticamente:

- `attr_reader :nombre` genera solo el getter (lectura). El dato se puede ver pero no cambiar desde fuera: ideal para cosas como id, fecha de creación, o saldos que solo cambian vía métodos con reglas (depositar/retirar, no asignación directa).
- `attr_writer :nombre` genera solo el setter (escritura). Raro en la práctica, pero existe para casos como passwords: los escribes pero no los lees.
- `attr_accessor :nombre` genera AMBOS (lectura y escritura). Para datos simples sin reglas, como el nombre de una persona o el título de un libro.

```

class Persona
  attr_reader :id          # solo lectura
  attr_accessor :nombre    # lectura y escritura
  attr_accessor :email     # lectura y escritura

  def initialize(id, nombre, email)
    @id = id
    @nombre = nombre
    @email = email
  end
end

p = Persona.new(1, "Ana", "ana@mail.com")
puts p.nombre    # "Ana" (attr_accessor permite leer)
p.nombre = "Anita"
puts p.nombre    # "Anita" (y escribir)
puts p.id        # 1 (attr_reader permite leer)
# p.id = 2       # NoMethodError: no hay setter para id

```

La decisión entre reader y accessor es una decisión de DISEÑO, no de flojera. Pregúntate: ¿tiene sentido que cualquiera cambie este dato directamente, sin reglas? El nombre de una persona: sí, cambio directo, accessor. El saldo de una cuenta: no, debe cambiar solo vía depositar/retirar que validan, así que reader (o ni eso). El id: nunca cambia, reader. Cada attr que escribes es una promesa sobre cómo se puede usar tu objeto; elige con intención.

Puedes declarar varios atributos en una línea: attr_accessor :nombre, :email, :telefono. Y los attr_ van normalmente al inicio de la clase, como una "ficha técnica" que anuncia qué datos expone el objeto. Cuando lees una clase ajena, lo primero que miras son sus attr_: te dicen de qué está hecho el objeto sin leer todo el código.

Detalle fino: attr_accessor genera métodos, y los métodos se pueden sobrecribir. Si necesitas un getter con lógica extra (por ejemplo, devolver el nombre capitalizado), defines el método a mano y no usas attr_reader para ese atributo:

```
class Persona
  def initialize(nombre)
    @nombre = nombre
  end

  def nombre
    @nombre.capitalize # getter personalizado con lógica
  end
end
```

Los atajos son para el caso simple; cuando necesites lógica, escribe el método. Ruby no te castiga por mezclar ambos estilos en la misma clase.

Errores comunes con clases y objetos: la galería de los horrores, parte 2

Error 1: Llamar métodos de instancia sobre la clase. La clase es el molde, no la galleta.

```
class Perro
  def ladrar
    puts "¡Guau!"
  end
end

# Código malo
Perro.ladrar # NoMethodError: undefined method 'ladrar' for Perro:Class

# Código bueno
f = Perro.new
f.ladrar # ¡Guau!
```

El mensaje de error te da la pista: "for Perro:Class" significa que se lo pediste a la CLASE. Los métodos de instancia se piden a OBJETOS creados con new. (Existen los métodos de clase, que sí se llaman sobre la clase, pero eso es tema avanzado; por ahora, si ves :Class en el error, crea el objeto primero.)

Error 2: olvidar el @ y crear una variable local fantasma.

```

class Contador
  def initialize
    total = 0 # ¡SIN @! Es una variable local que muere al terminar initialize
  end

  def incrementar
    total += 1 # NameError: aquí 'total' no existe
  end
end

# Código bueno: con @ la variable sobrevive entre métodos
class Contador
  def initialize
    @total = 0
  end

  def incrementar
    @total += 1
    @total
  end
end

```

Este error es traicionero porque no truena en initialize: la variable local se crea y muere en silencio. El trueno llega después, en otro método, lejos de la causa. Cuando un método no "recuerde" un valor que asignaste en otro, revisa los @.

Error 3: typo en el nombre de la variable de instancia. @nombre en un método y @nobre en otro: Ruby no avisa, el segundo vale nil.

```

# Código malo: lee @nomre (typo), obtiene nil sin error
def presentar
  puts "Me llamo #{@nomre}" # imprime "Me llamo " y nadie se entera del typo
end

```

Defensa: inicializa TODAS tus variables de instancia en initialize. Así, si un método usa una @ que no existe ahí, la sospecha cae rápido sobre el nombre. Algunos equipos usan herramientas de análisis que detectan esto, pero el hábito de "todo nace en initialize" ya te cubre el 90 por ciento.

Error 4: exponer todo con `attr_accessor` por flojera. Si el saldo de la cuenta tiene `accessor`, cualquiera puede hacer `cuenta.saldo = 1_000_000` y tus reglas de validación son papel mojado. Expón solo lo necesario: lo demás, `reader` o nada. El encapsulamiento no es paranoia, es la garantía de que tu objeto siempre está en un estado válido.

Error 5: meter TODA la lógica del programa en una sola clase gigante. Si tu clase tiene 40 métodos que hacen cosas sin relación (maneja usuarios, envía correos, genera PDFs, calcula impuestos), no es una clase: son cinco clases disfrazadas. Una clase debe representar UNA cosa del mundo real o del problema. Cuando los métodos dejen de tener que ver entre sí, es hora de dividir.

Error 6: no usar `initialize` y dejar objetos a medio construir. Ya lo vimos: si el objeto necesita datos para funcionar, que los pida al nacer. Un objeto que existe pero no sirve es una bomba de tiempo.

Buenas prácticas de POO en Ruby

Primero: modela cosas del mundo real o del problema, no cosas de la computadora. `Persona`, `CuentaBancaria`, `Pedido`, `Producto`, `Curso` son buenas clases. `ManejadorDeCosas`, `Utilidades`, `ProcesadorGeneral` son señales de que no entendiste el problema todavía. Si no puedes explicar en una frase qué representa tu clase, revísala.

Segundo: constructor honesto. `initialize` pide lo obligatorio, da defaults a lo opcional y valida lo crítico. Un objeto recién creado debe estar listo para usarse, sin pasos extra de configuración. Si crear el objeto requiere un manual de 5 pasos, el diseño está mal.

Tercero: esconde lo interno, muestra lo útil. Por defecto, todo es privado en tu mente: solo expones (con `attr_reader/accessor` o métodos públicos) lo que otros realmente necesitan. Es más fácil abrir después (agregar un `reader`) que cerrar (quitar un `accessor` que ya usa medio programa).

Cuarto: métodos que hablan el idioma del problema. `cuenta.retirar(100)` se entiende solo; `cuenta.actualizar_saldo_por_retiro(100)` es jerga de programador. Los métodos de un buen objeto suenan como las acciones que la cosa real haría: depositar, retirar, presentar, jugar, dormir, calificar.

Quinto: un objeto no debe saber demasiado de otros objetos. Si tu clase `Pedido` accede a `cliente.direccion.calle.numero_interior`, está metiendo la nariz muy adentro de otro objeto (esto se llama "envidia de atributos" y viola la ley de Demeter). Mejor: `cliente.direccion_completa`, un método que le pide al objeto lo que necesitas sin husmear en sus entrañas.

Sexto: prefiere composición sobre herencia complicada. En lugar de crear jerarquías profundas (Animal > Mamífero > Perro > PerroPolicia), compone objetos simples (un Perro TIENE una Raza, TIENE un Entrenador). La herencia es poderosa pero rígida; la composición es flexible. Para este nivel del curso, con clases simples y composición básica tienes de sobra.

Séptimo: prueba tus clases desde un script pequeño antes de integrarlas. Crea el objeto, llama cada método, prueba los casos borde (retirar más de lo que hay, crear sin datos). Cinco minutos de pruebas manuales te ahorran una hora de depuración después.

Composición: objetos que usan objetos

Hasta ahora nuestras clases han sido solitarias: un Perro, una CuentaBancaria, un Termómetro. Pero los programas reales son equipos de objetos que colaboran: un Pedido TIENE un Cliente y varios Productos; una Biblioteca TIENE Libros; un Carro TIENE un Motor y cuatro Llantas. Esta técnica se llama COMPOSICIÓN: construir objetos grandes a partir de objetos pequeños, como armar con bloques de construcción.

La composición es la alternativa sana a la herencia complicada. En lugar de decir "un PerroPolicia ES UN Perro con entrenamiento" (herencia), dices "un Perro TIENE un Entrenamiento" (composición). La diferencia práctica: con composición puedes cambiar las piezas en tiempo de ejecución (quitarle el entrenamiento, cambiarlo por otro), mientras que la herencia es rígida (el perro nació policía y morirá policía).

Mira un ejemplo completo: una clase Motor y una clase Carro que USA un motor.

```
class Motor
  attr_reader :caballos

  def initialize(caballos)
    @caballos = caballos
    @encendido = false
  end

  def encender
    @encendido = true
    "Motor de #{@caballos} caballos encendido"
  end

  def apagar
    @encendido = false
    "Motor apagado"
  end
end
```



```

    def encendido?
      @encendido
    end
  end

class Carro
  def initialize(modelo, caballos)
    @modelo = modelo
    @motor = Motor.new(caballos) # composición: el carro CREA su motor
  end

  def arrancar
    puts "#{@modelo}: #{@motor.encender}"
  end

  def detener
    puts "#{@modelo}: #{@motor.apagar}"
  end

  def ficha
    estado = @motor.encendido? ? "encendido" : "apagado"
    "#{@modelo}, motor #{estado}"
  end
end

mi_carro = Carro.new("Tsuru", 105)
mi_carro.arrancar # Tsuru: Motor de 105 caballos encendido
puts mi_carro.ficha
mi_carro.detener

```

Fíjate en lo que pasa: Carro no sabe CÓMO funciona un motor; solo sabe QUÉ le puede pedir (encender, apagar, encendido?). El motor cuida sus propios datos (@encendido es privado del motor). Cada clase tiene su responsabilidad clara: Motor maneja combustión, Carro maneja el concepto de vehículo. Si mañana cambias Motor por MotorElectrico con la misma interfaz (encender, apagar, encendido?), Carro sigue funcionando sin tocar una línea. Esa es la flexibilidad que la composición regala.

La otra forma de composición es por INYECCIÓN: en lugar de que el objeto cree sus piezas, se las PASAN ya creadas. Esto es aún más flexible porque quien usa el objeto decide qué piezas ponerle:

```

class Pedido
  def initialize(cliente, productos)
    @cliente = cliente      # objeto inyectado
    @productos = productos  # arreglo de objetos inyectado
  end

  def total
    @productos.sum { |p| p.precio }
  end

  def resumen
    lineas = @productos.map { |p| "- #{p.nombre}: ${p.precio}" }
    "Pedido de #{@cliente.nombre}:\n" + lineas.join("\n") + "\nTotal: ${total}"
  end
end

```

Aquí Pedido no crea al cliente ni a los productos: los recibe. Esto permite reutilizar el mismo cliente en varios pedidos y probar Pedido con clientes falsos. La regla general: si la pieza tiene vida propia fuera del objeto (un Cliente existe sin el Pedido), inyéctala; si la pieza solo existe como parte del objeto (el Motor del ejemplo anterior difícilmente vive solo), el objeto puede crearla.

Un error común al componer es romper el encapsulamiento de las piezas: en lugar de pedirle al objeto lo que necesitas, husmeas en sus entrañas.

```

# Código malo: el pedido husmea dentro del cliente
calle = pedido.cliente.direccion.calle # cadena de llamadas frágil

# Código bueno: el cliente ofrece lo que se necesita
calle = pedido.cliente.calle # un método del cliente, él sabe dónde vive el dato

```

Cada punto extra en la cadena es una dependencia extra: si Cliente cambia cómo guarda la dirección, todo el código que husmeaba se rompe. En cambio, si Cliente ofrece el método calle, puede reorganizar sus entrañas libremente. A esto se le llama la Ley de Demeter, y en lenguaje de la calle significa: no le pidas a un objeto que te presente a sus amigos para pedirles favores; pídele el favor a él.

Otro patrón útil con composición: objetos que se comportan como colecciones. La Biblioteca del ejercicio 10 es el ejemplo: guarda libros en un arreglo interno pero no regala el arreglo; ofrece agregar, buscar_por_autor, total y cada_libro. Quien usa la biblioteca no sabe (ni le importa) si adentro hay un arreglo, un hash o una base de datos. Si mañana cambias el arreglo por otra estructura, la interfaz no cambia y nadie se entera. Ese es el encapsulamiento a escala de sistema.

Para cerrar: cuando diseñes, pregúntate "¿esto ES UN ... o TIENE UN ...?". Un Perro ES UN Animal (herencia, úsala con moderación). Un Carro TIENE UN Motor, un Pedido TIENE Productos, una Biblioteca TIENE Libros (composición, tu herramienta favorita). La mayoría de los diseños buenos están hechos 90 por ciento de composición y 10 por ciento de herencia. Domina la composición y tus programas van a ser flexibles sin ser complicados.

Ejemplos paso a paso

Ejemplo 1: mi primer método con bloque y yield. El programa define un método cronometrar que mide cuánto tarda el bloque en ejecutarse. Muestra el patrón "sándwich": código antes, yield, código después.

```
def cronometrar(tarea)
  inicio = Time.now
  puts "Iniciando: #{tarea}"
  yield
  fin = Time.now
  puts "Terminó en #{(fin - inicio).round(3)} segundos"
end

cronometrar("suma pesada") do
  total = 0
  1_000_000.times { |i| total += i }
  puts "Total: #{total}"
end
```

El método envuelve tu bloque con medición de tiempo sin que tu bloque sepa nada de relojes. Este patrón (envolver con setup/teardown) es la base de cosas avanzadas como transacciones de base de datos y manejo de archivos.

Ejemplo 2: do/end vs llaves en acción. El mismo procesamiento escrito de las dos formas correctas, más un ejemplo del error clásico de precedencia comentado para que lo reconozcas cuando lo veas.

```

numeros = [1, 2, 3, 4, 5]

# Llaves: transformación de una línea, me importa el resultado
cuadrados = numeros.map { |n| n * n }
puts cuadrados.inspect # [1, 4, 9, 16, 25]

# do/end: procedimiento multilínea, me importan los efectos
numeros.each do |n|
  puts "El cuadrado de #{n} es #{n * n}"
end

# Forma segura de combinar puts con map y bloque
puts(numeros.select { |n| n.even? }.inspect) # [2, 4]

```

La regla de oro visual: ¿el bloque produce un valor que usas? Llaves. ¿El bloque hace cosas (imprime, guarda)? do/end. Y nunca pegues un do/end a un puts sin paréntesis.

Ejemplo 3: método que recibe bloque como proc con &. El programa muestra un método aplicar_dos_veces que ejecuta el bloque dos veces, recibéndolo como &bloque para poder llamarlo con .call múltiples veces.

```

def aplicar_dos_veces(&bloque)
  puts "Primera vez:"
  bloque.call
  puts "Segunda vez:"
  bloque.call
end

aplicar_dos_veces { puts "Hola desde el bloque" }

# También puedes pasar un proc guardado
despedida = Proc.new { puts "Adiós desde el proc" }
aplicar_dos_veces(&despedida)

```

Con &bloque el bloque se vuelve un objeto manipulable: lo llamas cuando quieres, cuantas veces quieras, o se lo pasas a otro método. Sin el &, solo tendrías yield (una ejecución por llamada en el punto donde lo pongas).

Ejemplo 4: procs y lambdas lado a lado. Un programa comparativo que muestra las dos diferencias clave: tolerancia de argumentos y comportamiento de return, con salida impresa para que veas el contraste.

```

puts "--- Argumentos ---"
p = Proc.new { |a, b| puts "proc: a=#{a.inspect} b=#{b.inspect}" }
p.call(1) # funciona: b es nil

l = ->(a, b) { puts "lambda: a=#{a.inspect} b=#{b.inspect}" }
l.call(1, 2) # funciona: argumentos exactos
# l.call(1) # ArgumentError si lo descomentas

puts "--- Return ---"
def con_proc
  pr = Proc.new { return "el proc escapó del método" }
  pr.call
  "esto nunca se imprime"
end

def con_lambda
  lam = -> { return "la lambda solo salió de sí misma" }
  resultado = lam.call
  "el método continúa: #{resultado}"
end

puts con_proc # el proc escapó del método
puts con_lambda # el método continúa: la lambda solo salió de sí misma

```

Memoriza el contraste: proc tolerante + return que escapa; lambda estricta + return local. Con eso decides correctamente en cada situación.

Ejemplo 5: clase Persona completa. La clase pedida en la tarea: con initialize, variables de instancia, métodos de instancia y attr_accessor. Incluye presentarse, cumpleaños (que incrementa la edad) y mayor_de_edad? (predicado con ?).

```

class Persona
  attr_accessor :nombre, :email
  attr_reader :edad

  def initialize(nombre, email, edad)
    @nombre = nombre
    @email = email
    @edad = edad
  end

  def presentarse
    "Hola, soy #{@nombre}, tengo #{@edad} años. Escribeme a #{@email}."
  end

  def cumpleanos
    @edad += 1
    "¡Feliz cumpleaños! Ahora tienes #{@edad}."
  end

  def mayor_de_edad?
    @edad >= 18
  end
end

ana = Persona.new("Ana", "ana@mail.com", 30)
puts ana.presentarse
puts ana.mayor_de_edad? # true
puts ana.cumpleanos     # 31

ana.nombre = "Anita"    # accessor permite escribir
puts ana.nombre         # "Anita"
# ana.edad = 25         # NoMethodError: edad es solo reader

```

Diseño intencional: nombre y email son accessor (cambian libremente), edad es reader (solo se modifica vía cumpleaños, con regla). mayor_de_edad? usa ? porque responde sí/no. Cada objeto guarda sus propios datos: crea dos personas y compruébalo.

Ejemplo 6: clase CuentaBancaria completa. La segunda clase pedida: titular obligatorio, saldo con default 0, depositar y retirar con validaciones, y transferencias entre cuentas. El ejemplo estrella del encapsulamiento.

```

class CuentaBancaria
  attr_reader :titular, :saldo

  def initialize(titular, saldo = 0)
    @titular = titular
    @saldo = saldo
  end

  def depositar(cantidad)
    return "Cantidad inválida" if cantidad <= 0
    @saldo += cantidad
    "Depósito ok. Saldo de #{@titular}: $#{@saldo}"
  end

  def retirar(cantidad)
    return "Cantidad inválida" if cantidad <= 0
    return "Fondos insuficientes" if cantidad > @saldo
    @saldo -= cantidad
    "Retiro ok. Saldo de #{@titular}: $#{@saldo}"
  end

  def transferir(destino, cantidad)
    resultado = retirar(cantidad)
    return resultado if resultado == "Fondos insuficientes" || resultado ==
"Cantidad inválida"
    destino.depositar(cantidad)
    "Transferencia ok de #{@titular} a #{destino.titular}"
  end
end

ana = CuentaBancaria.new("Ana", 1000)
luis = CuentaBancaria.new("Luis")

puts ana.depositar(500)      # Saldo: $1500
puts ana.retirar(2000)      # Fondos insuficientes
puts ana.transferir(luis, 300)
puts "Ana: $#{@ana.saldo}"  # 1200
puts "Luis: $#{@luis.saldo}" # 300
# ana.saldo = 999999        # NoMethodError: saldo es reader, no se puede trampear

```

transferir reutiliza retirar y depositar: no duplica la lógica de validación. Y como saldo es reader, nadie puede asignar el saldo a mano: la única forma de mover dinero es por los métodos con reglas. Eso es encapsulamiento protegiendo tu dinero (bueno, tu programa).

Ejemplo 7: bloque con parámetros y yield con valores. Un método `mi_each` reinventado para un arreglo, que demuestra cómo el método controla la iteración y el bloque decide la acción, pasando el elemento y el índice con `yield`.

```
def mi_each(arreglo)
  i = 0
  while i < arreglo.length
    yield(arreglo[i], i)
    i += 1
  end
end

mi_each(["a", "b", "c"]) do |letra, indice|
  puts "#{indice}: #{letra.upcase}"
end

# 0: A
# 1: B
# 2: C
```

Acabas de implementar `each_with_index` desde cero. El método maneja el `while` y el índice; el bloque recibe (elemento, índice) y decide qué hacer. Separación de responsabilidades perfecta, y ahora entiendes qué hacen por dentro los iteradores que usas a diario.

Ejemplo 8: menú de operaciones con hash de lambdas. Un programa interactivo que guarda operaciones matemáticas como lambdas en un hash y las ejecuta según la elección del usuario. Muestra callables como datos de primera clase.


```

ops = {
  "1" => { nombre: "Sumar", fn: ->(a, b) { a + b } },
  "2" => { nombre: "Restar", fn: ->(a, b) { a - b } },
  "3" => { nombre: "Multiplicar", fn: ->(a, b) { a * b } }
}

puts "Elige: 1) Sumar 2) Restar 3) Multiplicar"
opcion = gets.chomp

if ops.key?(opcion)
  puts "Dame dos números:"
  a = gets.chomp.to_f
  b = gets.chomp.to_f
  resultado = ops[opcion][:fn].call(a, b)
  puts "#{ops[opcion][:nombre]}: #{resultado}"
else
  puts "Opción inválida"
end

```

Sin lambdas, este menú sería una cadena de if/elsif repetitiva. Con el hash de lambdas, agregar una operación nueva es agregar una línea al hash: cero cambios en la lógica. Esa es la elegancia de tratar comportamiento como datos.

Ejemplo 9: clase Libro con bloque de configuración. Un ejemplo del patrón "bloque de configuración" usado por muchas gemas: el constructor recibe un bloque que se ejecuta en el contexto del objeto nuevo para configurarlo de forma expresiva.

```

class Libro
  attr_accessor :titulo, :autor, :paginas

  def initialize
    @paginas = 0
    yield(self) if block_given?
  end

  def descripcion
    "#{@titulo} de #{@autor} (#{@paginas} páginas)"
  end
end

libro = Libro.new do |l|
  l.titulo = "El Quijote"
  l.autor = "Cervantes"
  l.paginas = 863
end

puts libro.descripcion

```

`yield(self)` le pasa el objeto recién nacido al bloque, que lo configura. El resultado se lee como una receta: "un libro nuevo con título X, autor Y". Muchas gemas de Ruby usan exactamente este patrón para configuración elegante.

Ejemplo 10: programa integrador con todo lo de la unidad. Una clase Tienda que usa bloques para procesar ventas, procs para descuentos configurables y objetos Producto. El examen final de la unidad en forma de programa.

```

class Producto
  attr_accessor :nombre, :precio

  def initialize(nombre, precio)
    @nombre = nombre
    @precio = precio
  end
end

class Tienda
  def initialize
    @productos = []
    @descuento = ->(precio) { precio } # sin descuento por defecto
  end

  def agregar(producto)
    @productos << producto
  end

  def configurar_descuento(&bloque)
    @descuento = bloque # guardamos el bloque como proc
  end

  def ticket
    total = 0
    puts "--- Ticket ---"
    @productos.each do |p|
      final = @descuento.call(p.precio)
      puts "#{p.nombre}: ${p.precio} -> ${final.round(2)}"
      total += final
    end
    puts "TOTAL: ${total.round(2)}"
  end
end

tienda = Tienda.new
tienda.agregar(Producto.new("Café", 50))
tienda.agregar(Producto.new("Pan", 30))

tienda.configurar_descuento { |precio| precio * 0.9 } # 10% off
tienda.ticket

```

Este programa une bloques (configurar_descuento guarda el bloque), lambdas/procs (@descuento es un callable intercambiable), clases (Producto, Tienda), initialize, variables de

instancia y attr_accessor. Si lo lees de corrido y entiendes cada pieza, estás listo para construir programas orientados a objetos de verdad.

Práctica

Ejercicio 1: Define un método repetir(n) que ejecute el bloque n veces usando yield, sin usar times. Pruébalo imprimiendo "Hola" 3 veces.

Solución explicada:

```
def repetir(n)
  i = 0
  while i < n
    yield
    i += 1
  end
end

repetir(3) { puts "Hola" }
```

El método controla el ciclo con while y aprieta yield en cada vuelta; el bloque decide qué hacer. Es times reinventado: separas el "cuántas veces" (el método) del "qué hacer" (el bloque). Si llamas repetir(3) sin bloque, obtienes LocalJumpError, porque yield exige un bloque.

Ejercicio 2: Define un método con_bloque_seguro que ejecute el bloque solo si se pasó uno (usa block_given?); si no, imprima "Sin bloque, nada que hacer". Pruébalo de las dos formas.

Solución explicada:

```
def con_bloque_seguro
  if block_given?
    yield
  else
    puts "Sin bloque, nada que hacer"
  end
end

con_bloque_seguro { puts "Bloque ejecutado" }
con_bloque_seguro
```

`block_given?` pregunta si la llamada incluyó bloque. Este patrón hace tus métodos robustos: funcionan en modo simple (sin bloque) y en modo avanzado (con bloque), sin tronar nunca por un `yield` huérfano.

Ejercicio 3: Convierte estos bloques a la sintaxis convencional correcta (llaves o `do/end` según corresponda) y explica por qué: (a) `[1,2,3].map do |n| n * 2 end` usado dentro de `puts` en la misma línea; (b) un `each` multilínea que imprime y guarda, escrito con llaves.

Solución explicada:

```
# (a) Código original problemático:
# puts [1, 2, 3].map do |n| n * 2 end
# El do/end se pega a puts por precedencia, no a map. Corrección con llaves:
puts([1, 2, 3].map { |n| n * 2 }.inspect)

# (b) Código original con llaves multilínea (difícil de leer):
# [1, 2, 3].each { |n| puts "n=#{n}"; guardar(n) }
# Corrección con do/end:
[1, 2, 3].each do |n|
  puts "n=#{n}"
  # guardar(n)
end
```

La convención: llaves para transformaciones de una línea donde importa el valor devuelto (`map`, `select`); `do/end` para procedimientos multilínea con efectos secundarios (`each` que imprime o guarda). Y el detalle técnico: las llaves se asocian al método más cercano, `do/end` al más externo, así que con `puts` en la línea, usa llaves o paréntesis.

Ejercicio 4: Guarda en una variable `es_par` un `proc` que devuelva `true` si el número es par. Úsalo con `select` sobre `[1, 2, 3, 4, 5, 6]` (recuerda el `&`). Luego define una `lambda` `positivo?` con sintaxis `->` que devuelva `true` si el número es mayor que cero, y pruébala con `call`.

Solución explicada:

```
es_par = Proc.new { |n| n.even? }
puts [1, 2, 3, 4, 5, 6].select(&es_par).inspect # [2, 4, 6]

positivo? = ->(n) { n > 0 }
puts positivo?.call(5) # true
puts positivo?.call(-3) # false
```

El `&` convierte el `proc` en bloque para que `select` lo acepte. La `lambda` con `->` se comporta como un método miniatura: si la llamas con el número incorrecto de argumentos, truena con

ArgumentError, mientras que el proc sería tolerante. Para un predicado simple reutilizable, cualquiera sirve; la lambda avisa más fuerte si la usas mal.

Ejercicio 5: Demuestra con código la diferencia de return entre proc y lambda: define un método que cree un proc con return adentro y lo ejecute, y otro método igual pero con lambda. Imprime lo que devuelve cada método y explica la diferencia.

Solución explicada:

```
def metodo_con_proc
  p = Proc.new { return "el proc me sacó del método" }
  p.call
  "esta línea nunca se ejecuta"
end

def metodo_con_lambda
  l = -> { return "la lambda solo salió de sí misma" }
  texto = l.call
  "el método siguió y recibió: #{texto}"
end

puts metodo_con_proc
# el proc me sacó del método
puts metodo_con_lambda
# el método siguió y recibió: la lambda solo salió de sí misma
```

El return del proc escapa del método que lo contiene (como un botón de eyección), por eso la última línea de metodo_con_proc jamás corre. El return de la lambda es local: sale solo de la lambda y el método continúa. Por eso las lambdas son seguras para guardar y ejecutar después, y los procs con return solo son seguros si se ejecutan dentro del método original.

Ejercicio 6: Crea la clase Rectangulo con initialize que reciba base y altura, métodos area y perimetro, y un método cuadrado? (con ?) que devuelva true si base == altura. Crea dos objetos y prueba todo.

Solución explicada:

```

class Rectangulo
  def initialize(base, altura)
    @base = base
    @altura = altura
  end

  def area
    @base * @altura
  end

  def perimetro
    2 * (@base + @altura)
  end

  def cuadrado?
    @base == @altura
  end
end

r1 = Rectangulo.new(4, 5)
r2 = Rectangulo.new(3, 3)
puts r1.area          # 20
puts r1.perimetro     # 18
puts r1.cuadrado?     # false
puts r2.cuadrado?     # true

```

Cada objeto guarda su propia base y altura en variables de instancia; los métodos las usan sin necesidad de parámetros. `cuadrado?` lleva ? porque responde sí/no. Nota que `r1` y `r2` no se interfieren: el `@base` de `r1` es independiente del de `r2`.

Ejercicio 7: A la clase `Rectangulo` del ejercicio anterior agrégle `attr_reader` para `base` y `attr_accessor` para `altura`. Demuestra que puedes leer ambas, cambiar la altura, pero no cambiar la base directamente.

Solución explicada:

```

class Rectangulo
  attr_reader :base
  attr_accessor :altura

  def initialize(base, altura)
    @base = base
    @altura = altura
  end

  def area
    @base * @altura
  end
end

r = Rectangulo.new(4, 5)
puts r.base      # 4 (lectura ok)
puts r.altura    # 5 (lectura ok)
r.altura = 10
puts r.area      # 40 (cambió la altura)
# r.base = 7     # NoMethodError: base es solo reader

```

La decisión de diseño: la altura se puede ajustar libremente (accessor), pero la base es de solo lectura (reader). Si alguien intenta `r.base = 7`, Ruby lanza `NoMethodError` porque no existe el setter. Así proteges la integridad del objeto según tus reglas.

Ejercicio 8: Crea la clase `Termometro` con `initialize` que reciba `grados_celsius` (default 20), un método `leer` que devuelva los grados, un método `subir(grados)` y `bajar(grados)` que no permitan salir del rango -50 a 60 (si se pasa, lo topa al límite e informa). Usa `attr_reader` para los grados.

Solución explicada:


```

class Termometro
  attr_reader :grados

  def initialize(grados_celsius = 20)
    @grados = grados_celsius
  end

  def subir(cantidad)
    @grados += cantidad
    if @grados > 60
      @grados = 60
      "¡Tope máximo! Se quedó en 60"
    else
      "Ahora: #{@grados}"
    end
  end

  def bajar(cantidad)
    @grados -= cantidad
    if @grados < -50
      @grados = -50
      "¡Tope mínimo! Se quedó en -50"
    else
      "Ahora: #{@grados}"
    end
  end
end

t = Termometro.new
puts t.subir(50) # ¡Tope máximo! Se quedó en 60
puts t.grados   # 60
puts t.bajar(200) # ¡Tope mínimo! Se quedó en -50

```

Los grados solo se leen desde fuera (reader); solo subir y bajar los modifican, y siempre respetando los límites. Nadie puede hacer `t.grados = 999` porque no hay setter. Las reglas del termómetro viven dentro del termómetro: encapsulamiento puro.

Ejercicio 9: Define un método `procesar(lista, &filtro)` que aplique el bloque filtro a cada elemento con `select` y devuelva los que pasen, pero si no se pasa bloque devuelva la lista completa. Pruébalo con y sin bloque.

Solución explicada:

```

def procesar(lista, &filtro)
  return lista unless filtro
  lista.select(&filtro)
end

puts procesar([1, 2, 3, 4]).inspect          # [1, 2, 3, 4] (sin bloque)
puts procesar([1, 2, 3, 4]) { |n| n.even? }.inspect  # [2, 4] (con bloque)

pares = Proc.new { |n| n.even? }
puts procesar([1, 2, 3, 4], &pares).inspect        # [2, 4] (con proc)

```

Con &filtro el bloque llega como proc: si es nil (no pasaron bloque), devolvemos la lista tal cual; si existe, lo convertimos de vuelta a bloque con & para select. Este patrón (bloque opcional como filtro) aparece en APIs reales que ofrecen un comportamiento por defecto ampliable.

Ejercicio 10: Diseña la clase Biblioteca que guarde libros (usa la clase Libro del Ejemplo 9 o un hash simple con título y autor). Debe tener: agregar(libro), buscar_por_autor(autor) que devuelva los libros coincidentes, total (cantidad de libros) y un método cada_libro que reciba un bloque y lo ejecute por cada libro (usa yield). Pruébala con 3 libros.

Solución explicada:

```

class Libro
  attr_accessor :titulo, :autor

  def initialize(titulo, autor)
    @titulo = titulo
    @autor = autor
  end
end

class Biblioteca
  def initialize
    @libros = []
  end

  def agregar(libro)
    @libros << libro
  end

  def buscar_por_autor(autor)
    @libros.select { |l| l.autor == autor }
  end

  def total
    @libros.length
  end

  def cada_libro
    @libros.each { |l| yield(l) }
  end
end

biblio = Biblioteca.new
biblio.agregar(Libro.new("Cien años de soledad", "García Márquez"))
biblio.agregar(Libro.new("El amor en los tiempos del cólera", "García Márquez"))
biblio.agregar(Libro.new("El Quijote", "Cervantes"))

puts biblio.total # 3
puts biblio.buscar_por_autor("García Márquez").length # 2

biblio.cada_libro do |libro|
  puts "- #{libro.titulo}"
end

```

Este ejercicio integra todo: dos clases colaborando (Biblioteca USA objetos Libro: composición), variables de instancia, métodos que devuelven valores útiles, y `cada_libro` que expone la colección interna mediante `yield` sin regalar el arreglo `@libros` (nadie puede modificarlo directamente desde fuera). Si llegaste hasta aquí entendiendo cada línea, la unidad está dominada.

Ejercicio 11: Crea una clase `Dado` con `initialize` que no reciba nada e inicie `@caras` en 6, un método `lanzar` que devuelva un número aleatorio entre 1 y `@caras` (usa `rand(1..@caras)`), y un método `lanzar_varias(vuces)` que reciba un bloque y lo ejecute por cada lanzamiento pasándole el resultado con `yield`. Pruébalo lanzando 5 veces e imprimiendo cada resultado.

Solución explicada:

```
class Dado
  def initialize
    @caras = 6
  end

  def lanzar
    rand(1..@caras)
  end

  def lanzar_varias(vuces)
    veces.times do
      yield(lanzar)
    end
  end
end

d = Dado.new
d.lanzar_varias(5) do |resultado|
  puts "Salió: #{resultado}"
end
```

`lanzar_varias` combina POO con bloques: el objeto controla la repetición (`times`) y el bloque decide qué hacer con cada resultado, que viaja por `yield`. Es el mismo patrón que `each`: el método itera, el bloque actúa. Nota que `rand(1..@caras)` usa la variable de instancia, así que si mañana agregas un dado de 20 caras, solo cambias `@caras`.

Ejercicio 12: Diseña las clases `Autor` y `Libro2` donde `Libro2` recibe un objeto `Autor` en su `initialize` (composición por inyección). `Autor` tiene `nombre` y `nacionalidad` (`attr_reader` para ambos). `Libro2` tiene `titulo` y `autor` (`attr_reader`), y un método `ficha` que devuelva "Titulo, por Nombre (Nacionalidad)". Crea un autor con dos libros y muestra sus fichas.

Solución explicada:

```
class Autor
  attr_reader :nombre, :nacionalidad

  def initialize(nombre, nacionalidad)
    @nombre = nombre
    @nacionalidad = nacionalidad
  end
end

class Libro2
  attr_reader :titulo, :autor

  def initialize(titulo, autor)
    @titulo = titulo
    @autor = autor # composición: el libro TIENE un autor (objeto)
  end

  def ficha
    "#{@titulo}, por #{@autor.nombre} (#{@autor.nacionalidad})"
  end
end

gabo = Autor.new("García Márquez", "colombiana")
l1 = Libro2.new("Cien años de soledad", gabo)
l2 = Libro2.new("El amor en los tiempos del cólera", gabo)

puts l1.ficha # Cien años de soledad, por García Márquez (colombiana)
puts l2.ficha
```

El mismo objeto gabo se comparte entre los dos libros: si corriges el nombre del autor en un solo lugar (reconstruyendo el objeto), ambos libros lo reflejan. Libro2 no husmea en las entrañas de Autor: usa sus métodos públicos nombre y nacionalidad. Esto es composición por inyección bien hecha: piezas independientes que colaboran por interfaces claras.

Ejercicio 13: Crea la clase Estudiante con nombre (attr_accessor), una variable de instancia @calificaciones que inicie como hash vacío, un método agregar_calificacion(materia, valor) que valide que valor esté entre 0 y 10, un método promedio que use los valores del hash, un predicado aprobado? (promedio >= 6) y un método reporte que reciba un bloque y se lo pase a cada par materia-calificacion con yield. Pruébalo todo.

Solución explicada:

```

class Estudiante
  attr_accessor :nombre

  def initialize(nombre)
    @nombre = nombre
    @calificaciones = {}
  end

  def agregar_calificacion(materia, valor)
    return "Calificación inválida" unless valor.between?(0, 10)
    @calificaciones[materia] = valor
    "Registrada: #{materia} = #{valor}"
  end

  def promedio
    return 0 if @calificaciones.empty?
    @calificaciones.values.sum / @calificaciones.length.to_f
  end

  def aprobado?
    promedio >= 6
  end

  def reporte
    @calificaciones.each { |materia, valor| yield(materia, valor) }
  end
end

e = Estudiante.new("Paco")
puts e.agregar_calificacion(:mate, 8)
puts e.agregar_calificacion(:fisica, 15) # Calificación inválida
puts e.agregar_calificacion(:historia, 7)
puts e.promedio          # 7.5
puts e.aprobado?         # true
e.reporte do |materia, calif|
  puts "#{materia}: #{calif}"
end

```

Esta clase es el compendio de las dos unidades: un hash como estructura interna (@calificaciones), métodos con validación y return temprano, retorno implícito, predicado con ?, bloque con yield que expone los datos sin regalar el hash, y between? como validación elegante de rango. El objeto se cuida solo: nadie puede meter un 15 ni leer el hash directamente; todo pasa por la ventanilla de métodos. Si escribiste algo parecido sin mirar, ya piensas en objetos.

Resumen de la unidad

- Un bloque es código sin nombre que pasas a un método: el método decide cuándo ejecutarlo. Sintaxis: do/end (multilínea, efectos) y {} (una línea, transformaciones).
- yield ejecuta el bloque dentro del método; puede recibir argumentos que llegan a los parámetros del bloque (|x|). block_given? pregunta si hubo bloque.
- Cuidado con la precedencia: {} se pega al método más cercano, do/end al más externo. Con puts en la misma línea, usa llaves o paréntesis.
- Un proc (Proc.new) es un bloque embotellado: lo guardas en variable y lo ejecutas con call. Con & lo conviertes a bloque (&proc) y con &bloque en la definición capturas el bloque como proc.
- Una lambda (->(x) { ... }) es como un método anónimo. Diferencias con proc: la lambda es estricta con los argumentos (ArgumentError si no cuadran) y su return sale solo de la lambda; el proc es tolerante y su return escapa del método que lo contiene.
- POO: junta datos y comportamiento en objetos. La clase es el molde (class/end, nombre en CamelCase), el objeto es la instancia (Clase.new).
- initialize es el constructor: Ruby lo llama automáticamente con los argumentos de new. Pide lo obligatorio, da defaults a lo opcional, valida lo crítico.
- Las variables de instancia (@) son privadas de cada objeto y viven entre llamadas a métodos. Inicialízalas todas en initialize y cuida los typos (una @ mal escrita vale nil en silencio).
- Los métodos de instancia (def dentro de class) son lo que el objeto sabe hacer; se llaman con objeto.metodo. Diseña métodos que hablen el idioma del problema y validen antes de actuar.
- attr_reader genera getter (lectura), attr_writer genera setter (escritura), attr_accessor genera ambos. Elige según el diseño: lo que cualquiera puede cambiar sin reglas lleva accessor; lo que debe cambiar solo con validación lleva reader o nada.

- Errores clásicos: llamar métodos de instancia sobre la clase, olvidar el @, tipos en variables de instancia, exponer todo con accessor por flojera, y clases gigantes que hacen de todo.
- Buenas prácticas: modela cosas reales, constructor honesto, esconde lo interno, métodos con nombres del problema, no husmees en otros objetos, prefiere composición, y prueba cada clase en un script pequeño antes de integrarla.
- La composición construye objetos grandes a partir de objetos pequeños (un Carro TIENE un Motor, un Pedido TIENE Productos): es más flexible que la herencia complicada. Decide entre "el objeto crea sus piezas" y "se las inyectan" según si la pieza tiene vida propia fuera de él, y respeta la Ley de Demeter: pídele favores al objeto, no a los amigos del objeto.

Para cerrar esta unidad, quédate con la imagen completa: los bloques te enseñaron a pasar comportamiento como si fuera un dato más, y los objetos te enseñaron a unir datos con comportamiento en unidades que se cuidan solas. Esas dos ideas son el corazón de Ruby: por eso el código Ruby se lee tan expresivo (bloques por todas partes) y por eso escala tan bien (objetos que encapsulan su complejidad). Ya no piensas en "variables y funciones sueltas": piensas en objetos que colaboran, cada uno con su responsabilidad, comunicándose por mensajes. Con las seis unidades que llevas, tienes el lenguaje completo para construir programas serios: estructuras de datos, control de flujo, hashes, métodos, bloques y objetos. Lo que sigue en tu camino es práctica, práctica y más práctica: toma cualquier programa procedural que hayas escrito antes y reescríbelo orientado a objetos. Vas a sorprenderte de cuánto más claro queda.

UNIDAD 7: Herencia, módulos y strings

Bienvenido a la Unidad 7 del Curso Completo de Ruby de Lucas EGA Academy. Si llegaste hasta aquí, ya pasaste por variables, métodos, clases, arreglos, hashes y bloques. Ya sabes crear objetos, organizar datos y darle comportamiento a tus clases. Ahora toca el siguiente nivel: aprender a organizar ese conocimiento para que no repitas código, para que tus programas sean elegantes y para que puedas usar las herramientas más poderosas que Ruby te ofrece.

Esta unidad junta tres temas que, aunque parecen distintos, tienen algo en común: los tres tratan de escribir menos código y hacer más. La herencia te permite crear clases que nacen con todo el conocimiento de otra clase, como hijos que heredan las habilidades de sus papás y luego aprenden las suyas propias. Los módulos te permiten compartir comportamiento entre clases que no están relacionadas entre sí, como recetas de cocina que cualquiera puede usar aunque cocine cosas distintas. Y los strings avanzados te dan el poder de manipular texto como un profesional: interpolar valores, transformar mayúsculas, reemplazar palabras, limpiar espacios y manejar textos gigantes sin morir en el intento.

Ruby es famoso por ser un lenguaje donde "todo es un objeto", y en esta unidad vas a sentir esa filosofía en carne propia. Los strings dejan de ser simples letras y se convierten en herramientas con decenas de métodos útiles. Los módulos te muestran que puedes construir comportamientos reutilizables sin atarte a una jerarquía rígida de clases. Y la herencia te enseña el equilibrio entre compartir código y mantener cada clase con su propia personalidad.

Al terminar esta unidad vas a poder diseñar jerarquías de clases limpias (como Vehiculo, Auto y Moto), usar mixins famosos como Comparable y Enumerable para darle superpoderes a tus clases con apenas dos líneas, organizar tu código con namespaces para que no se peleen los nombres, y manipular texto con la confianza de quien tiene 30 métodos en el bolsillo. Así que afloja los hombros, abre tu editor y vamos a construir de verdad.

Explicación detallada

Qué es la herencia y por qué existe

Imagina que trabajas en una concesionaria y tienes que registrar vehículos. Primero creas la clase Vehiculo con marca, modelo, año y un método para encender el motor. Luego necesitas registrar autos: los autos también tienen marca, modelo, año y se encienden... ¿vas a copiar y pegar todo? Después llegan las motos: también marca, modelo, año y encendido. Y luego los

camiones. Si copias el mismo código en cada clase, terminas con cuatro versiones de lo mismo, y cuando encuentres un error en el método de encender el motor, tendrás que arreglarlo cuatro veces. Eso es una pesadilla de mantenimiento.

La herencia resuelve esto con una idea sencilla: una clase puede declarar que es hija de otra clase, y automáticamente recibe todos sus métodos y comportamientos. La clase hija (o subclase) hereda de la clase padre (o superclase). Es como en la vida real: un hijo hereda los rasgos de sus papás, pero además desarrolla su personalidad propia. Un Auto hereda todo lo de Vehiculo y le agrega sus cosas particulares, como el número de puertas o el método para abrir la cajuela.

En Ruby, la herencia se escribe con el símbolo menor que (<). La sintaxis es directa:

```
class Vehiculo
  def encender
    puts "Motor encendido"
  end
end

class Auto < Vehiculo
  def abrir_cajuela
    puts "Cajuela abierta"
  end
end

mi_auto = Auto.new
mi_auto.encender      # funciona, lo heredó de Vehiculo
mi_auto.abrir_cajuela # funciona, es suyo propio
```

Fíjate en algo importante: la clase Auto nunca definió el método encender, pero el objeto lo entiende. Eso es la herencia en acción: el objeto busca primero en su clase y, si no encuentra el método, sube a la clase padre a buscarlo. Ruby hace esto automáticamente con lo que se llama la cadena de ancestros (ancestors chain), que veremos con más detalle más adelante.

El error más común de los principiantes es pensar que la herencia sirve para "meter todo en la clase padre" o para relacionar clases que no tienen una relación de verdad. La regla de oro es la prueba del "es un": un Auto ES UN Vehiculo, por eso hereda de Vehiculo. Una Moto ES UN Vehiculo. Pero una Llanta NO ES UN Vehiculo, una llanta es PARTE de un vehículo. Para las partes se usa composición (el objeto tiene una llanta como atributo), no herencia. Si alguna vez dudas, pregúntate: "¿X es un Y?" Si la respuesta es sí, herencia. Si la respuesta es "X tiene un Y" o "X usa un Y", composición.

Otro error típico es crear jerarquías demasiado profundas: Vehiculo < VehiculoTerrestre < VehiculoConMotor < Auto < AutoDeportivo... Cada nivel agrega complejidad y hace el código difícil de seguir. Una buena práctica es mantener las jerarquías cortas: dos o tres niveles como máximo en la mayoría de los casos. Si sientes que necesitas más niveles, probablemente un módulo (que veremos en esta misma unidad) sea mejor solución.

El operador < y cómo funciona la cadena de ancestros

Cuando escribes `class Auto < Vehiculo`, le estás diciendo a Ruby: "la clase Auto tiene como padre a la clase Vehiculo". Pero lo que pasa por dentro es más interesante: Ruby construye una cadena de búsqueda de métodos. Cuando llamas `mi_auto.encender`, Ruby busca el método en este orden: primero en la clase del objeto (Auto), luego en los módulos incluidos en Auto (hablaremos de eso), luego en la clase padre (Vehiculo), luego en los módulos de Vehiculo, luego en la abuela (Object, porque todas las clases heredan de Object), y así hasta llegar a BasicObject.

Puedes ver esta cadena con el método `ancestors`:

```
class Vehiculo; end
class Auto < Vehiculo; end

puts Auto.ancestors.inspect
# [Auto, Vehiculo, Object, Kernel, BasicObject]
```

Esta cadena explica por qué todos los objetos entienden métodos como `puts`, `to_s` o `inspect` aunque nunca los defines: los heredan de Object y Kernel. Es herencia trabajando silenciosamente desde el día uno.

Un detalle que confunde a muchos: en Ruby solo existe herencia simple, es decir, una clase tiene UN SOLO padre directo. No puedes escribir `class Auto < Vehiculo, Maquina` porque eso es un error de sintaxis. Otros lenguajes permiten herencia múltiple, pero Ruby decidió que trae más problemas que beneficios (el famoso "problema del diamante", donde dos padres definen el mismo método y no se sabe cuál usar). Para compartir comportamiento entre clases no relacionadas, Ruby inventó los módulos y los mixins, que son más flexibles y no tienen ese problema. Así que no te preocupes: la herencia simple más los módulos cubren todo lo que necesitas.

Código malo (intentar herencia múltiple, esto ni siquiera corre):

```
# ESTO ES ERROR DE SINTAXIS EN RUBY
class Auto < Vehiculo, Maquina
end
```

Código bueno (una sola herencia + módulos para lo demás):

```
class Vehiculo
end

module Electrico
  def cargar_bateria
    puts "Cargando bateria..."
  end
end

class AutoElectrico < Vehiculo
  include Electrico
end
```

super: cómo llamar al método del papá

Aquí viene una de las herramientas más elegantes de Ruby. Cuando una clase hija define un método con el mismo nombre que uno del padre, el de la hija "tapa" al del padre. Eso se llama sobreescritura (override) y es intencional: la hija quiere comportarse diferente. Pero muchas veces no quieres reemplazar por completo el comportamiento del padre, sino extenderlo: hacer lo del padre MÁS algo extra. Para eso existe `super`.

`super` llama al método del mismo nombre en la clase padre. Es como decir: "oye papá, haz lo tuyo primero, y luego yo agrego lo mío". Mira este ejemplo clásico:

```

class Vehiculo
  def descripcion
    "Vehiculo generico"
  end
end

class Auto < Vehiculo
  def descripcion
    super + " con 4 puertas"
  end
end

puts Auto.new.descripcion
# "Vehiculo generico con 4 puertas"

```

Sin `super`, el método del padre se perdería por completo. Con `super`, lo reutilizas. El error de principiante más común es reescribir todo el código del padre dentro del hijo en lugar de llamar a `super`:

Código malo (duplicación):

```

class Vehiculo
  def initialize(marca, modelo)
    @marca = marca
    @modelo = modelo
  end
end

class Auto < Vehiculo
  def initialize(marca, modelo, puertas)
    @marca = marca      # copiado del padre, mal
    @modelo = modelo    # copiado del padre, mal
    @puertas = puertas
  end
end

```

Código bueno (con super):

```

class Vehiculo
  def initialize(marca, modelo)
    @marca = marca
    @modelo = modelo
  end
end

class Auto < Vehiculo
  def initialize(marca, modelo, puertas)
    super(marca, modelo) # el padre hace su parte
    @puertas = puertas   # el hijo agrega lo suyo
  end
end

```

Hay un detalle sutil con `super` que causa dolores de cabeza: `super` sin paréntesis pasa AUTOMÁTICAMENTE todos los argumentos que recibió el método hijo. `super()` con paréntesis vacíos no pasa NINGÚN argumento. Y `super(a, b)` pasa exactamente esos argumentos. Mira:

```

class Padre
  def saludar(nombre, edad)
    puts "Hola #{nombre}, tienes #{edad}"
  end
end

class Hijo < Padre
  def saludar(nombre, edad)
    super          # pasa nombre y edad automáticamente
  end
end

class Hijo2 < Padre
  def saludar(nombre, edad)
    super()        # NO pasa nada: ERROR, faltan argumentos
  end
end

class Hijo3 < Padre
  def saludar(nombre, edad)
    super(nombre)  # pasa solo nombre: ERROR, falta edad
  end
end

```

La regla práctica: si el método hijo recibe los mismos argumentos que el padre, usa ``super`` a secas (sin paréntesis). Si necesitas controlar qué se pasa, usa ``super`` con paréntesis y los argumentos explícitos. El error clásico es escribir ``super()`` pensando que es lo mismo que ``super``, y luego no entender por qué el padre dice que le faltan argumentos.

Sobreescritura de métodos: darle personalidad a la hija

La sobreescritura (override) es cuando la clase hija define un método que ya existe en el padre, cambiando su comportamiento. Es una de las ideas más poderosas de la programación orientada a objetos, porque permite algo llamado polimorfismo: tratar objetos distintos de la misma manera y que cada uno responda a su estilo.

Piensa en un estacionamiento que cobra por hora. Tiene una lista de vehículos y a cada uno le pregunta cuánto debe pagar. Un Auto paga una tarifa, una Moto paga menos, un Camión paga más. Gracias a la sobreescritura, el código que cobra no necesita saber qué tipo de vehículo es cada uno: simplemente llama al método ``tarifa`` y cada clase responde con su propia versión.

```

class Vehiculo
  def tarifa_por_hora
    50
  end
end

class Auto < Vehiculo
  def tarifa_por_hora
    60
  end
end

class Moto < Vehiculo
  def tarifa_por_hora
    30
  end
end

class Camion < Vehiculo
  def tarifa_por_hora
    120
  end
end

vehiculos = [Auto.new, Moto.new, Camion.new, Vehiculo.new]
vehiculos.each do |v|
  puts "#{v.class}: ${v.tarifa_por_hora} por hora"
end

```

El método `each` no sabe ni le importa qué clase es cada objeto. Cada uno responde `tarifa\_por\_hora` a su manera. Eso es polimorfismo, y la sobreescritura es lo que lo hace posible.

Pero ojo, hay una trampa: la sobreescritura debe respetar el "contrato" del método original. Si el método del padre promete devolver un número, el de la hija también debe devolver un número, no un string ni nil de la nada. Este principio se conoce como sustitución de Liskov, pero no necesitas memorizar el nombre: solo recuerda que una hija debe poder usarse donde se esperaba al padre sin romper nada. Si tu clase Auto rompe algo que funcionaba con Vehiculo, tu sobreescritura está mal diseñada.

Error común de principiante: sobrecribir un método cambiando su firma (los argumentos que recibe):

Código malo:

```
class Vehiculo
  def mover(destino)
    puts "Moviendose a #{destino}"
  end
end

class Auto < Vehiculo
  def mover # quito el argumento: rompo el contrato
    puts "El auto se mueve"
  end
end

# Quien use Auto.new.mover("casa") esperando el comportamiento
# del padre, se estrella con un ArgumentError
```

Código bueno:

```
class Auto < Vehiculo
  def mover(destino)
    super # reutilizo lo del padre
    puts "a 100 km/h" # agrego lo mio
  end
end
```

Jerarquía de ejemplo: Vehiculo, Auto y Moto paso a paso

Vamos a construir una jerarquía completa y realista para que veas cómo se juntan todas las piezas: atributos heredados, `initialize` con `super`, métodos sobrescritos y métodos propios de cada hija. Este es el tipo de diseño que harías en un proyecto de verdad.

```

class Vehiculo
  attr_reader :marca, :modelo, :anio

  def initialize(marca, modelo, anio)
    @marca = marca
    @modelo = modelo
    @anio = anio
    @encendido = false
  end

  def encender
    @encendido = true
    puts "#{@marca} #{@modelo}: motor encendido"
  end

  def apagar
    @encendido = false
    puts "#{@marca} #{@modelo}: motor apagado"
  end

  def encendido?
    @encendido
  end

  def descripcion
    "#{@anio} #{@marca} #{@modelo}"
  end

  def tarifa_por_hora
    50
  end
end

```

Esta es la base. Todo vehículo tiene marca, modelo, año, se puede encender y apagar, y tiene una tarifa base. Ahora las hijas:

```

class Auto < Vehiculo
  def initialize(marca, modelo, anio, puertas)
    super(marca, modelo, anio)
    @puertas = puertas
  end

  def descripcion
    super + " (#{@puertas} puertas)"
  end

  def tarifa_por_hora
    60
  end

  def abrir_cajuela
    puts "Cajuela del #{@modelo} abierta"
  end
end

class Moto < Vehiculo
  def initialize(marca, modelo, anio, cilindrada)
    super(marca, modelo, anio)
    @cilindrada = cilindrada
  end

  def descripcion
    super + " (#{@cilindrada}cc)"
  end

  def tarifa_por_hora
    30
  end

  def hacer_caballito
    puts "La #{@modelo} hace un caballito!"
  end
end

```

Fíjate en el patrón que se repite en cada hija: el `initialize` llama a `super` con los argumentos del padre y luego inicializa lo propio; `descripcion` llama a `super` y le agrega su detalle; `tarifa_por_hora` se sobrescribe por completo porque la tarifa no tiene nada que ver con la del padre; y cada hija tiene sus métodos únicos (`abrir_cajuela`, `hacer_caballito`).

Y ahora el uso polimórfico, que es donde brilla el diseño:

```

flota = [
  Auto.new("Nissan", "Versa", 2022, 4),
  Moto.new("Honda", "CB190", 2023, 190),
  Auto.new("Toyota", "Corolla", 2021, 4)
]

flota.each do |v|
  v.encender
  puts v.descripcion
  puts "Tarifa: ${v.tarifa_por_hora}/hora"
  puts "---"
end

```

Cada objeto responde con su propia versión de `descripcion` y `tarifa\_por\_hora`, pero todos entienden `encender` gracias a la herencia. Este es el diseño limpio al que debes aspirar: el padre pone lo común, las hijas ponen lo particular, y el código que usa los objetos no necesita preguntar de qué clase son.

Una buena práctica final sobre herencia: prefiere los métodos pequeños en el padre para que las hijas puedan sobrescribir pedacitos en lugar de métodos gigantes. Si el método `encender` del padre hiciera 50 cosas, la hija que quiera cambiar solo una tendría que copiar las otras 49 o llamar a `super` y aguantarse. Métodos pequeños y enfocados hacen la herencia flexible.

Qué es un módulo: el concepto

Si la herencia es la relación "es un" (un Auto es un Vehículo), los módulos son la respuesta a "puede hacer" o "se comporta como". Un módulo en Ruby es como una caja de herramientas o una receta: un paquete de métodos (y constantes) que puedes compartir entre clases que no tienen nada que ver entre sí.

Piensa en la habilidad de "ser comparable": puedes comparar números ($3 < 5$), strings ("a" < "b"), fechas y hasta tus propias clases si defines cómo compararlas. Los números y los strings no están relacionados por herencia, pero ambos "pueden compararse". En Ruby, esa capacidad compartida vive en un módulo llamado Comparable. Lo mismo pasa con Enumerable: arreglos, hashes y rangos "pueden recorrerse", y ese comportamiento vive en el módulo Enumerable.

Un módulo se define con la palabra `module` y, a diferencia de una clase, NO se puede instanciar: no puedes hacer `MiModulo.new`. Los módulos no crean objetos; existen para organizar código y para compartirse. Es como la diferencia entre un molde para galletas (la

clase, que sí produce galletas/objetos) y un recetario (el módulo, que solo guarda recetas/métodos para que otros los usen).

```
module Saludable
  def tomar_agua
    puts "Tomando 2 litros de agua"
  end

  def dormir_bien
    puts "Durmiendo 8 horas"
  end
end

class Persona
  include Saludable
end

class Atleta
  include Saludable
end

Persona.new.tomar_agua # funciona
Atleta.new.dormir_bien # funciona
```

Persona y Atleta no están relacionadas por herencia, pero ambas incluyen el módulo Saludable y obtienen sus métodos. A esto se le llama mixin: mezclar un módulo dentro de una clase. Es la forma rubyista de compartir comportamiento sin forzar una relación padre-hijo artificial.

El error de principiante aquí es usar herencia cuando lo correcto es un módulo. Si tienes clases Perro, Gato y Pez y las tres pueden "nadar" (bueno, el pez nada, el perro nada, el gato... digamos que flota), no tiene sentido crear una clase padre Nadador de la que hereden, porque un Perro no ES UN Nadador, un perro es un animal QUE PUEDE nadar. La relación correcta es un módulo Nadable incluido en cada clase. Pregúntate siempre: ¿es un...? (herencia) o ¿puede...? / ¿se comporta como...? (módulo).

Mixins: include vs extend, la diferencia que todo el mundo confunde

Aquí está el tema que más confunde a quienes aprenden Ruby, así que vamos despacio. Cuando metes un módulo en una clase hay dos formas, y hacen cosas distintas:

- ``include``: agrega los métodos del módulo como métodos DE INSTANCIA. Es decir, los objetos creados con la clase podrán usarlos.
- ``extend``: agrega los métodos del módulo como métodos DE CLASE. Es decir, la clase misma (no sus objetos) podrá usarlos.

Analogía: imagina que el módulo es una caja de herramientas. ``include`` le da una copia de la caja a cada trabajador (cada objeto). ``extend`` le da la caja al jefe (la clase misma). Los trabajadores usan sus herramientas; el jefe usa las suyas. Son copias separadas para niveles distintos.

```
module Utilidades
  def gritar(texto)
    puts texto.upcase + "!!!"
  end
end

class Persona
  include Utilidades # métodos de instancia
end

class Empresa
  extend Utilidades # métodos de clase
end

Persona.new.gritar("hola") # funciona: "HOLA!!!"
Empresa.gritar("hola")     # funciona: "HOLA!!!"

# Pero al revés NO funciona:
# Persona.gritar("hola")     # ERROR: método de instancia, no de clase
# Empresa.new.gritar("hola") # ERROR: método de clase, no de instancia
```

El 90% de las veces que uses mixins vas a usar ``include``, porque normalmente quieres que los OBJETOS tengan el comportamiento. ``extend`` se usa en casos más especiales, como cuando quieres darle métodos extra a una clase para configuración, o cuando quieres extender un objeto individual (sí, también puedes hacer ``mi_objeto.extend(MiModulo)`` para darle métodos solo a ese objeto, pero eso ya es tema avanzado).

Error típico: usar ``extend`` cuando querías ``include`` y luego no entender por qué tus objetos "no encuentran" el método:

Código malo (confusión):

```

module Volador
  def volar
    puts "Volando..."
  end
end

class Pajaro
  extend Volador # el programador quería que los pájaros vuelen
end

Pajaro.new.volar # ERROR: undefined method 'volar'
# Funciona Pajaro.volar, pero eso no tiene sentido:
# la CLASE Pajaro no vuela, vuelan los pájaros (los objetos)

```

Código bueno:

```

class Pajaro
  include Volador # ahora cada pájaro (objeto) puede volar
end

Pajaro.new.volar # "Volando..."

```

Regla para no equivocarte nunca: pregúntate "¿quién debe poder llamar a este método, el objeto o la clase?" Si la respuesta es el objeto (casi siempre), usa `include`. Memotécnica: **include = instancias**.

Comparable y Enumerable: los mixins famosos

Ruby incluye en su biblioteca estándar dos módulos que son la mejor publicidad de los mixins: Comparable y Enumerable. Conocerlos te ahorra escribir decenas de métodos a mano.

Comparable te da todos los operadores de comparación (<, >, <=, >=, ==, !=) y métodos como `between?` con solo definir UN método: el operador nave espacial `<=>`. Este operador devuelve -1 si el objeto es menor, 0 si es igual, 1 si es mayor, y nil si no se pueden comparar. Es como enseñarle a Ruby tu criterio de "quién va primero".

```

class Jugador
  include Comparable
  attr_reader :nombre, :puntos

  def initialize(nombre, puntos)
    @nombre = nombre
    @puntos = puntos
  end

  def <=>(otro)
    @puntos <=> otro.puntos
  end
end

ana = Jugador.new("Ana", 150)
beto = Jugador.new("Beto", 200)

puts ana < beto          # true (150 < 200)
puts beto > ana          # true
puts ana.between?(Jugador.new("X", 100), Jugador.new("Y", 180)) # true
puts [beto, ana].sort.map(&:nombre).inspect # ["Ana", "Beto"], sort funciona
gratis

```

Con un solo método (`<=>`) obtuviste seis operadores, `between?` y la capacidad de ordenarte con `sort`. Eso es el poder de un mixin bien diseñado. El error común es incluir Comparable pero olvidar definir `<=>`, o definirlo devolviendo true/false en lugar de -1/0/1. Si `<=>` devuelve true/false, las comparaciones se comportan raro o fallan: recuerda, no es una pregunta de sí/no, es una brújula de tres direcciones.

Enumerable es todavía más impresionante: te da `map`, `select`, `reject`, `find`, `count`, `sort`, `min`, `max`, `sum`, `each\_with\_index`, `group\_by` y docenas más, con solo definir UN método: `each`. Si tu clase sabe recorrerse con `each`, Enumerable le regala todo un arsenal.


```

class Equipo
  include Enumerable
  def initialize(*jugadores)
    @jugadores = jugadores
  end

  def each(&bloque)
    @jugadores.each(&bloque)
  end
end

equipo = Equipo.new("Ana", "Beto", "Carla")
puts equipo.count # 3
puts equipo.map { |j| j.upcase }.inspect # ["ANA", "BETO", "CARLA"]
puts equipo.select { |j| j.start_with?("A") }.inspect # ["Ana"]
puts equipo.include?("Beto") # true

```

La lección es profunda: los arreglos y hashes que usas todos los días no definen `map` y `select` cada uno por su cuenta; ambos incluyen Enumerable y solo definen su propio `each`. Cuando entiendes esto, entiendes cómo está construido Ruby por dentro. Y cuando diseñes tus propias colecciones (una Baraja de cartas, un Inventario, una ListaDeTareas), ya sabes el truco: define `each`, incluye Enumerable, y listo.

Buena práctica: antes de escribir a mano métodos como `filtrar`, `buscar` o `contar` en una clase colección, pregúntate si incluyendo Enumerable y definiendo `each` los obtienes gratis. Nueve de cada diez veces, sí.

Namespaces: organizar código con módulos

Los módulos tienen un segundo superpoder además de los mixins: sirven como namespaces, es decir, como apellidos para tu código. Imagina que en tu programa tienes una clase `Reporte` para ventas y otra clase `Reporte` para inventario. Si las dos se llaman igual, chocan. La solución es meter cada una en un módulo con el nombre de su área:

```

module Ventas
  class Reporte
    def generar
      puts "Reporte de ventas generado"
    end
  end
end

module Inventario
  class Reporte
    def generar
      puts "Reporte de inventario generado"
    end
  end
end

Ventas::Reporte.new.generar      # "Reporte de ventas generado"
Inventario::Reporte.new.generar  # "Reporte de inventario generado"

```

El operador `::` es como decir el nombre completo: "el Reporte de Ventas" vs "el Reporte de Inventario". Es exactamente igual que en la vida real: puede haber dos personas que se llamen Juan, pero "Juan Pérez" y "Juan García" no se confunden.

Esto es especialmente útil cuando usas bibliotecas de terceros: si una gema define una clase `Cliente` y tú también tienes una clase `Cliente`, el namespace evita el choque. Por eso las gemas serias envuelven todo su código en un módulo con su nombre.

Los módulos también guardan constantes, y el namespace aplica igual:

```

module Configuracion
  VERSION = "2.1.0"
  MAX_INTENTOS = 3
end

puts Configuracion::VERSION      # "2.1.0"
puts Configuracion::MAX_INTENTOS # 3

```

Error común: olvidar el namespace al referirse a la clase desde fuera del módulo, o usarlo cuando ya estás dentro. Dentro del módulo puedes usar el nombre corto; fuera, necesitas el completo:

Código malo:

```
module Ventas
  class Reporte; end
end

Reporte.new # ERROR: no existe Reporte "a secas" en este contexto
```

Código bueno:

```
Ventas::Reporte.new # correcto desde fuera

module Ventas
  class Factura
    def crear
      Reporte.new # correcto desde DENTRO: ya estoy en el namespace
    end
  end
end
```

Buena práctica: cuando tu programa crezca y tenga más de unas 5-6 clases, empieza a agruparlas en módulos por tema (Ventas, Usuarios, Reportes). Tu yo del futuro te lo va a agradecer cuando el proyecto tenga 50 archivos.

Strings: de letras sueltas a herramientas profesionales

En Ruby, los strings no son ciudadanos de segunda clase: son objetos con más de 100 métodos. Si vienes de otros lenguajes donde un string es casi un dato tonto, aquí te vas a sorprender. Un string en Ruby sabe ponerse en mayúsculas, limpiarse, reemplazarse partes, dividirse, medirse, preguntar si está vacío y hasta congelarse. Esta sección te enseña lo esencial para trabajar como profesional.

Primero, un recordatorio importante que ya viste en unidades anteriores pero que vale la pena fijar: en Ruby hay dos formas de escribir strings, con comillas dobles (" ") y con comillas simples (' '). La diferencia es que las comillas dobles INTERPRETAN interpolación y secuencias de escape (\n, \t), mientras que las simples las toman literal:

```
nombre = "Ana"
puts "Hola #{nombre}, bienvenida\nNos vemos" # interpreta todo
puts 'Hola #{nombre}, bienvenida\nNos vemos' # literal, sale tal cual
```

Regla práctica: usa comillas dobles cuando necesites interpolación o saltos de línea; usa simples para texto plano. Muchos programadores usan dobles por defecto para no pensarlo dos veces, y está bien.

Interpolación #{}

La interpolación es meter el resultado de una expresión dentro de un string usando #{ }. Es la forma más legible y más rápida de construir texto dinámico, y es una de las señas de identidad de Ruby.

```
nombre = "Carlos"
edad = 25
puts "Me llamo #{nombre} y tengo #{edad} años"
# Dentro de #{ } puede ir CUALQUIER expresión Ruby:
puts "El próximo año tendré #{edad + 1} años"
puts "Mi nombre en mayúsculas es #{nombre.upcase}"
puts "Hoy es #{Time.now.strftime('%Y-%m-%d')}"
```

El error de principiante es concatenar con + en lugar de interpolar:

Código malo:

```
puts "Me llamo " + nombre + " y tengo " + edad.to_s + " años"
# feo, largo, y truena si olvidas el .to_s en edad
```

Código bueno:

```
puts "Me llamo #{nombre} y tengo #{edad} años"
# limpio, y Ruby convierte todo a string solo
```

La interpolación llama automáticamente a `to_s` en el resultado, así que no necesitas conversiones manuales. Además es más rápida que concatenar con + en la mayoría de los casos, porque evita crear strings intermedios. Úsala siempre que puedas: tu código se lee como una frase, no como un rompecabezas.

Un truco avanzado pero útil: puedes interpolar variables de instancia y llamar métodos directamente: `"#{@nombre}"`, `"#{usuario.nombre}"`. Y si necesitas un # literal seguido de llaves en el texto, escápalo con backslash: `"#{no se interpola}"`.

Los métodos de string que vas a usar el 95% del tiempo

Memoriza estos; son los que aparecen en casi todos los programas:

Cambiar mayúsculas y minúsculas: ``upcase``, ``downcase``, ``capitalize``, ``swapcase``.

```
texto = "hola Mundo"
puts texto.upcase      # "HOLA MUNDO"
puts texto.downcase    # "hola mundo"
puts texto.capitalize  # "Hola mundo"
puts texto.swapcase    # "HOLA mUNDO"
```

Ojo: estos métodos devuelven un string NUEVO, no modifican el original. Si quieres modificar el original, usa la versión con signo de exclamación: ``upcase!``, ``downcase!``, etc. Esta convención del ! significa "versión peligrosa que modifica el receptor" y aplica a muchos métodos de Ruby:

```
texto = "hola"
texto.upcase
puts texto  # "hola" (no cambió, upcase devolvió uno nuevo que nadie guardó)

texto.upcase!
puts texto  # "HOLA" (ahora sí cambió)
```

Error clásico: llamar ``texto.upcase`` esperando que ``texto`` cambie, y luego usar ``texto`` como si estuviera en mayúsculas. Si no guardas el resultado o no usas !, el cambio se pierde.

Limpiar espacios: ``strip``, ``lstrip``, ``rstrip``. QUITAN espacios (y saltos de línea, tabs) del inicio y final, o solo de un lado:

```
entrada = "  ana@gmail.com  \n"
puts entrada.strip.inspect  # "ana@gmail.com"
puts entrada.lstrip.inspect # "ana@gmail.com  \n"
puts entrada.rstrip.inspect # "  ana@gmail.com"
```

Esto es ORO cuando lees datos del usuario con ``gets``, porque ``gets`` incluye el salto de línea del Enter. El patrón ``gets.chomp`` que ya conoces es primo de estos: ``chomp`` quita solo el salto de línea final. Para formularios y validaciones, ``strip`` es tu mejor amigo: un email con espacios al final no debe fallar tu validación.

Medir y preguntar: ``length`` (o su alias ``size``), ``empty?``, ``include?``, ``start_with?``, ``end_with?``:

```

texto = "programar en Ruby"
puts texto.length      # 17
puts texto.empty?      # false
puts texto.include?("Ruby") # true
puts texto.start_with?("pro") # true
puts texto.end_with?("Ruby") # true
puts "".empty?         # true

```

Dividir y unir: ``split`` convierte un string en arreglo, ``join`` hace lo contrario (es método de Array pero vive en pareja con `split``):

```

csv = "ana,beto,carla"
nombres = csv.split(",")
puts nombres.inspect # ["ana", "beto", "carla"]

puts nombres.join(" - ") # "ana - beto - carla"

```

``split`` sin argumentos divide por espacios en blanco (y colapsa múltiples espacios), perfecto para procesar frases:

```

frase = "Ruby es genial"
puts frase.split.inspect # ["Ruby", "es", "genial"]

```

Reemplazar: ``sub`` reemplaza la PRIMERA ocurrencia, ``gsub`` reemplaza TODAS (la g es de "global"):

```

texto = "el gato come y el gato duerme"
puts texto.sub("gato", "perro") # "el perro come y el gato duerme"
puts texto.gsub("gato", "perro") # "el perro come y el perro duerme"

```

Error común: usar ``sub`` esperando que cambie todas las ocurrencias. Si quieres todas, es ``gsub``. Y recuerda: sin !, devuelven un string nuevo.

gsub con expresiones regulares simples

Aquí es donde los strings se vuelven superpoderosos. Una expresión regular (regex) es un patrón de búsqueda: en lugar de buscar el texto exacto "gato", buscas "cualquier dígito" o "cualquier palabra que empiece con vocal". En Ruby las regex se escriben entre diagonales: `/patron/`.

No necesitas ser experto en regex para sacarle provecho. Con estos patrones básicos resuelves el 80% de los casos:

- `\d` : cualquier dígito (0-9)
- `\w` : cualquier letra, dígito o guion bajo
- `\s` : cualquier espacio en blanco
- `.` : cualquier caracter (uno solo)
- `+` : "una o más veces" (aplicado a lo anterior)
- `*` : "cero o más veces"

```
texto = "Mi teléfono es 555-1234 y mi otro es 555-9876"
puts texto.gsub(/\d/, "X")
# "Mi teléfono es XXX-XXXX y mi otro es XXX-XXXX"

# Limpiar un texto dejando solo letras y espacios:
sucio = "Hola!!! ¿Cómo estás??? 123"
limpio = sucio.gsub(/[^a-zA-Záéíóúñü\s]/, "")
puts limpio # "Hola Cómo estás "
```

Ese último ejemplo merece explicación: `[^...]` significa "cualquier caracter que NO esté en esta lista". Entonces `/[^a-zA-Záéíóúñü\s]/` encuentra todo lo que no sea letra (incluyendo acentos y ñ) ni espacio, y `gsub` lo reemplaza con nada (""). Es la receta estándar para sanitizar texto.

Otro uso cotidiano: normalizar espacios múltiples:

```
texto = "Ruby    es    genial"
puts texto.gsub(/\s+/, " ") # "Ruby es genial"
```

`\s+` significa "uno o más espacios seguidos", y los reemplazamos por un solo espacio. Simple y elegante.

También puedes CAPTURAR partes del patrón con paréntesis y reutilizarlas con `\1`, `\2` en el reemplazo:

```
fecha = "2026-09-19"
# Reordenar a formato 19/09/2026:
puts fecha.gsub(/(\d+)-(\d+)-(\d+)/, '\3/\2/\1')
# "19/09/2026"
```

Aquí (\d+) captura "uno o más dígitos" tres veces, y en el reemplazo \3 es la tercera captura (el día), \2 la segunda (el mes), \1 la primera (el año). Ojo con las comillas simples en el reemplazo: si usaras dobles, Ruby intentaría interpolar \1 como variable. Con regex, usa comillas simples para el string de reemplazo cuando uses \1, \2.

Buena práctica: cuando una regex se pone complicada, no la escribas de memoria a la primera; pruébala en un script pequeño con varios ejemplos antes de meterla a tu programa. Las regex son poderosas pero traicioneras: un patrón mal escrito puede borrar más de lo que querías.

Heredoc: textos largos sin pelear con comillas

Cuando necesitas un texto de varias líneas (una plantilla de email, un mensaje largo, un bloque de ayuda), escapar saltos de línea con \n se vuelve ilegible. Para eso existe el heredoc: escribes el texto tal cual, en varias líneas, delimitado por una marca:

```
mensaje = <<~MENSAJE
  Hola #{nombre}:

  Gracias por registrarte en Lucas EGA Academy.
  Tu curso de Ruby ya está disponible.

  Saludos,
  El equipo
MENSAJE

puts mensaje
```

La sintaxis <<~MENSAJE significa: "todo lo que sigue, hasta la línea que diga MENSAJE, es un string". El ~ (squiggly heredoc, disponible desde Ruby 2.3) además quita la indentación común, así tu código se ve limpio. La marca puede ser cualquier palabra en mayúsculas por convención.

Hay tres variantes que debes conocer:

- <<~ : quita la indentación (la que más vas a usar)
- <<- : permite indentar la marca de cierre pero NO quita la indentación del texto
- << : la marca de cierre debe ir al inicio de la línea (la forma vieja)

Y como con las comillas, hay versión que interpola y versión literal:


```

nombre = "Ana"
# Con comillas en la marca, NO interpola:
literal = <<~'TEXT0'
    Hola #{nombre}, esto sale tal cual
TEXT0
puts literal # "Hola #{nombre}, esto sale tal cual"

```

Error común: olvidar que la marca de cierre debe estar SOLA en su línea, sin nada más (con <<~ puede ir indentada, pero nada de texto extra). Otro error: elegir una marca que aparece dentro del texto; si tu texto contiene la palabra MENSAJE en una línea sola, el heredoc termina antes de tiempo. Usa marcas descriptivas y poco comunes como EMAIL_BIENVENIDA o AYUDA_TEXTO.

El heredoc brilla en casos reales: plantillas de correo, mensajes de ayuda de un programa de consola, consultas SQL largas, o el texto de un reporte. Compáralo con la alternativa:

Código malo (concatenación infernal):

```

email = "Hola " + nombre + ":\n\n" +
        "Gracias por registrarte.\n" +
        "Tu curso ya está disponible.\n\n" +
        "Saludos,\nEl equipo\n"

```

Código bueno (heredoc):

```

email = <<~EMAIL
    Hola #{nombre}:

    Gracias por registrarte.
    Tu curso ya está disponible.

    Saludos,
    El equipo
EMAIL

```

No hay discusión: el heredoc se lee como el documento final, no como código.

Congelar strings: freeze y por qué importa

En Ruby, los strings son mutables por defecto: puedes cambiarlos con `<<`, `gsub!`, `upcase!`, etc. Eso es flexible, pero tiene un costo: cada vez que escribes "hola" en tu código, Ruby crea un objeto string nuevo en memoria. En un programa que repite un millón de veces una comparación con el mismo string, eso es un millón de objetos basura.

Para eso existe `freeze`: congela un objeto para que no pueda modificarse. Si intentas modificar un string congelado, Ruby lanza un error `FrozenError`. Suena agresivo, pero es una protección: los valores que no deben cambiar (constantes, claves de configuración, mensajes fijos) deberían estar congelados para que ningún bug los modifique por accidente.

```
SALUDO = "Hola".freeze
puts SALUDO # "Hola"

# SALUDO.upcase! # ERROR: FrozenError, no se puede modificar
SALUDO = "Adiós" # esto sí se puede: reasignar la constante (con warning)
```

Fíjate en la sutileza: `freeze` congela el OBJETO (el string "Hola"), no la variable. Reasignar la constante a otro string es otra operación (Ruby te muestra una advertencia pero lo permite).

El uso más común en código real es congelar strings que se usan como claves o que se repiten mucho:

```
# Sin freeze: cada iteración crea un string nuevo para "procesando"
1000000.times { puts "procesando".object_id }

# Con freeze: un solo objeto reutilizado
PROCESANDO = "procesando".freeze
1000000.times { puts PROCESANDO.object_id } # mismo id siempre
```

Desde Ruby 3.0, los strings que escribes literalmente en el código con cierta configuración pueden congelarse automáticamente (magic comment ``# frozen_string_literal: true`` al inicio del archivo), que es lo que usan las aplicaciones grandes como Rails para ahorrar memoria. No necesitas usarlo en tus ejercicios, pero ahora sabes que existe y por qué.

Buena práctica: congela las constantes string de tu programa. Es gratis, previene bugs sutiles (un método que modifica sin querer un mensaje compartido) y en programas grandes ahorra memoria. Piensa en freeze como ponerle un candado a lo que no debe cambiar.

Errores comunes con strings (y cómo evitarlos)

Antes de pasar a los ejemplos, un repaso de las trampas donde más caen los principiantes con strings:

1. **Comparar con == cuando el caso importa:** "Ana" == "ana" es false. Si comparas nombres de usuario o emails, normaliza primero: ``email1.downcase == email2.downcase`` o mejor ``email1.casecmp?(email2)`` (compara ignorando mayúsculas, devuelve true/false).
2. **Olvidar que gets trae el salto de línea:** ``nombre = gets`` guarda "Ana\n". Siempre usa ``gets.chomp`` (o ``gets.strip`` si también quieres quitar espacios).
3. **Confundir length con el número de "cosas":** ``"hola".length`` es 5 (caracteres). Para contar palabras, es ``"hola mundo".split.length`` (2). Son preguntas distintas.
4. **Interpolación en comillas simples:** ``"Hola #{nombre}"`` sale literal. Si necesitas interpolación, comillas dobles o heredoc sin comillas en la marca.
5. **Modificar el string que no era:** métodos sin ! devuelven uno nuevo; si no lo guardas, el cambio se pierde. Decide siempre: ¿quiero modificar el original (usa !) o crear uno nuevo (guarda el resultado)?
6. **Sumar strings y números:** ``"Edad: " + 25`` truena con TypeError. Usa interpolación: ``"Edad: #{25}"``.

Herencia vs módulos: la guía definitiva de decisión

Ya viste herencia y módulos por separado, pero la pregunta que todo programador Ruby se hace es: ¿cuándo uso cada uno? La respuesta cabe en una frase: **herencia para lo que ES, módulos para lo que PUEDE HACER**. Pero en la práctica hay matices, así que aquí tienes la guía completa con ejemplos de decisiones reales.

Usa HERENCIA cuando:

- Hay una relación "es un" clara y permanente: un Perro es un Animal, un Auto es un Vehículo.
- Las hijas comparten ESTADO (atributos) además de comportamiento: todas tienen @marca, @modelo.
- Quieres que el polimorfismo funcione con una familia cerrada de clases que tú controlas.

Usa MÓDULOS (mixins) cuando:

- Varias clases no relacionadas comparten un COMPORTAMIENTO: Cliente, Empleado y Proveedor pueden ser "corteseros".

- Quieres agregar una capacidad opcional: Volador, Nadable, Descontable, Comparable.
- Necesitas compartir comportamiento entre clases que YA tienen padres distintos (recuerda: solo hay un padre por clase, pero puedes incluir muchos módulos).

Mira un diseño que combina ambos, como lo harías en un proyecto real:

```
class Animal
  attr_reader :nombre

  def initialize(nombre)
    @nombre = nombre
  end

  def dormir
    puts "#{@nombre} está durmiendo"
  end
end

module Volador
  def volar
    puts "#{@nombre} está volando"
  end
end

module Nadador
  def nadar
    puts "#{@nombre} está nadando"
  end
end

class Perro < Animal
  include Nadador # los perros nadan

  def ladrar
    puts "#{@nombre}: ¡Guau!"
  end
end

class Pato < Animal
  include Volador # los patos vuelan...
  include Nadador # ...y también nadan

  def graznar
    puts "#{@nombre}: ¡Cuac!"
  end
end
```

```

end

class Pez < Animal
  include Nadador

  def burbujear
    puts "#{@nombre}: glub glub"
  end
end

[Perro.new("Firulais"), Pato.new("Lucas"), Pez.new("Nemo")].each do |a|
  a.dormir # todos: viene de Animal (herencia)
end

Pato.new("Lucas").volar # solo el pato: viene del módulo
Perro.new("Firulais").nadar # perro y pato y pez: módulo compartido

```

Fíjate en lo expresivo del diseño: la herencia pone la base común (todo animal tiene nombre y duerme), los módulos agregan capacidades que cruzan las fronteras de la familia (el pato vuela y nada, el pez solo nada, el perro solo nada). Intentar modelar esto SOLO con herencia te obligaría a crear clases intermedias absurdas como AnimalVoladorYNadador. Los módulos evitan esa explosión de clases.

El antipatrón a evitar es la "herencia por conveniencia": crear una clase padre solo para no copiar dos métodos, aunque la relación "es un" no exista. Ejemplo real que se ve mucho: `class ReportePDF < ConexionBaseDeDatos` porque "necesito los métodos de conexión". Eso es un abuso: un reporte NO ES una conexión. Lo correcto es que Reporte USE una conexión (composición) o incluya un módulo Conectable. La herencia por conveniencia crea acoplamiento raro: si mañana cambias la conexión, se rompe el reporte sin razón.

Preguntas de autoevaluación antes de decidir:

1. ¿Puedo decir "X es un Y" sin que suene raro? Sí -> herencia. No -> sigue.
2. ¿Varias clases necesitan este comportamiento y no comparten padre? Sí -> módulo.
3. ¿Es una capacidad opcional que algunas instancias querrían y otras no? -> módulo.
4. ¿Comparten atributos y ciclo de vida? -> herencia (o composición si es "tiene un").

Encadenamiento de métodos: el estilo fluido de Ruby

Una de las bellezas de Ruby es que casi todos los métodos de string devuelven un string nuevo, lo que permite ENCADENARLOS: el resultado de uno alimenta al siguiente, como una línea de producción. En lugar de variables temporales, escribes una sola expresión que se lee de izquierda a derecha.

```
# Sin encadenar: variables temporales, más líneas, más ruido
texto = " aNA LÓPEZ "
paso1 = texto.strip
paso2 = paso1.downcase
paso3 = paso2.split
paso4 = paso3.map(&:capitalize)
resultado = paso4.join(" ")
puts resultado # "Ana López"

# Encadenado: una sola expresión, se lee como receta
resultado = " aNA LÓPEZ ".strip.downcase.split.map(&:capitalize).join(" ")
puts resultado # "Ana López"
```

La versión encadenada se lee como una receta de cocina: "quita espacios, pasa a minúsculas, divide en palabras, capitaliza cada una, únelas con espacio". Cada método devuelve un string (o arreglo) y el siguiente lo toma. Este estilo es idiomático en Ruby y lo verás en código profesional por todas partes.

Pero hay reglas para encadenar bien:

1. **Cuida el tipo que devuelve cada eslabón.** ``split`` devuelve un ARREGLO, no un string: después de split solo puedes llamar métodos de arreglo (map, select, join). Si intentas ``"a b".split.upcase``, truena, porque Array no tiene upcase. Conoce qué devuelve cada método.
2. **No encadenes demasiado.** Tres a cinco eslabones se leen bien; diez se vuelven un acertijo. Si la cadena es muy larga, dale nombre a los pasos intermedios con variables bien nombradas.
3. **Cuidado con los métodos ! en cadenas.** ``" hola ".strip.upcase!`` modifica el string que devolvió strip (uno nuevo, no hay problema), pero ``"hola".upcase!.downcase`` también funciona. El peligro es cuando el receptor es una variable que quieres conservar: ``texto.strip!`` modifica texto para siempre. En cadenas, prefiere las versiones sin ! salvo que quieras modificar a propósito.

Un ejemplo encadenado de la vida real, limpiando un título para usarlo como nombre de archivo (slug):

```

titulo = " ¡Mi Primer Post: Ruby es Genial! "
slug = titulo.downcase # " ¡mi primer post: ruby es genial! "
      .gsub(/^[^a-z0-9\s]/, "") # " mi primer post ruby es genial "
      .strip # "mi primer post ruby es genial"
      .gsub(/\s+/, "-") # "mi-primer-post-ruby-es-genial"

puts slug

```

(Ruby permite partir la cadena en varias líneas después del punto: cada línea empieza con . y se lee perfecto.) Este slug es exactamente lo que hacen los blogs para convertir títulos en URLs amigables. Cuatro eslabones, cero variables temporales, resultado profesional.

Extraer datos de strings: match, scan y el operador []

Hasta ahora transformaste strings; ahora vas a EXTRACTAR información de ellos, que es lo que haces cuando procesas logs, analizas texto o parseas respuestas. Ruby te da tres herramientas principales.

El operador []: extrae por posición o por patrón. Es el más simple:

```

texto = "Ruby es genial"
puts texto[0] # "R" (primer caracter)
puts texto[0, 4] # "Ruby" (desde 0, toma 4)
puts texto[-6, 6] # "genial" (índices negativos cuentan desde el final)
puts texto["genial"] # "genial" (si contiene el substring, lo devuelve)
puts texto[/\d+/] # nil (no hay dígitos; con regex devuelve la coincidencia)

```

match: busca un patrón regex y devuelve un objeto MatchData con la coincidencia y las capturas:

```

linea = "Pedido #4521 por $1,250.00"
m = linea.match(/#(\d+) por \$(.+)/)
if m
  puts "Folio: #{m[1]}" # "4521"
  puts "Monto: #{m[2]}" # "1,250.00"
  puts "Completo: #{m[0]}" # "#4521 por $1,250.00"
end

```

MatchData es como un parte policiaco de la coincidencia: m[0] es todo lo que coincidió, m[1], m[2] son las capturas entre paréntesis. Si no hay coincidencia, match devuelve nil, por eso el if: intentar m[1] sobre nil truena con NoMethodError.

scan: encuentra TODAS las coincidencias y las devuelve en un arreglo. Perfecto para extraer listas de cosas:

```
texto = "Llama al 555-1234 o al 555-9876, o escribe a ana@correo.com"
telefonos = texto.scan(/\d{3}-\d{4}/)
puts telefonos.inspect # ["555-1234", "555-9876"]

emails = texto.scan(/[\\w.]+@[\\w.]+\\.\\w+/)
puts emails.inspect # ["ana@correo.com"]

# Con paréntesis de captura, scan devuelve arreglo de arreglos:
datos = "ana:25, beto:30".scan(/(\\w+):(\\d+)/)
puts datos.inspect # ["ana", "25"], ["beto", "30"]]
```

La diferencia entre match y scan es como la diferencia entre encontrar LA llave perdida y encontrar TODAS las monedas en el sillón: match se detiene en la primera coincidencia (y te da detalle), scan recorre todo el texto (y te da la lista). Para validar ("¿cumple el formato?") usa match?; para extraer una cosa usa match; para extraer muchas usa scan.

Un ejemplo completo que junta todo: parsear una línea de log de servidor:

```
log = '192.168.1.10 - - [19/Sep/2026:10:30:00 -0600] "GET /cursos/ruby HTTP/1.1" 200 4521'

m = log.match(/(\\S+) .* \\.\\(\\.+?)\\) "(\\w+) (\\S+).*" (\\d+) (\\d+)/)
if m
  puts "IP:      #{m[1]}"
  puts "Fecha:   #{m[2]}"
  puts "Método:  #{m[3]}"
  puts "Ruta:     #{m[4]}"
  puts "Código:   #{m[5]}"
  puts "Bytes:    #{m[6]}"
end
```

Los logs de servidores reales tienen exactamente este formato, y parsearlos con una regex así es una tarea cotidiana en administración de sistemas. Cada (\\S+) o (\\.+?) captura un pedazo, y MatchData te los entrega ordenados. No necesitas entender cada símbolo de la regex al 100% hoy: lo importante es el patrón de trabajo (match + capturas + if contra nil).

Ejemplos paso a paso

Ejemplo 1: Jerarquía Vehículo, Auto y Moto completa

Este es el ejemplo central de la unidad: una jerarquía funcional que usa herencia, super, sobreescritura y polimorfismo. Cópialo, ejecútalo y modifícalo: agrega una clase Camion o una clase BicicletaElectrica para practicar.

```
class Vehiculo
  attr_reader :marca, :modelo, :anio

  def initialize(marca, modelo, anio)
    @marca = marca
    @modelo = modelo
    @anio = anio
    @encendido = false
  end

  def encender
    @encendido = true
    puts "#{descripcion}: motor encendido"
  end

  def apagar
    @encendido = false
    puts "#{descripcion}: motor apagado"
  end

  def descripcion
    "#{@anio} #{@marca} #{@modelo}"
  end

  def tarifa_por_hora
    50
  end
end

class Auto < Vehiculo
  def initialize(marca, modelo, anio, puertas)
    super(marca, modelo, anio)
    @puertas = puertas
  end

  def descripcion
    super + " (#{@puertas} puertas)"
  end
end
```

```

def tarifa_por_hora
  60
end

def abrir_cajuela
  puts "Cajuela abierta del #{@modelo}"
end
end

class Moto < Vehiculo
  def initialize(marca, modelo, anio, cilindrada)
    super(marca, modelo, anio)
    @cilindrada = cilindrada
  end

  def descripcion
    super + " (#{@cilindrada}cc)"
  end

  def tarifa_por_hora
    30
  end

  def hacer_caballito
    puts "La #{@modelo} hace un caballito!"
  end
end

# Uso polimórfico: el mismo código funciona con cualquier vehículo
flota = [
  Auto.new("Nissan", "Versa", 2022, 4),
  Moto.new("Honda", "CB190", 2023, 190),
  Auto.new("Toyota", "Corolla", 2021, 4),
  Moto.new("Yamaha", "FZ", 2020, 150)
]

total = 0
flota.each do |v|
  v.encender
  puts v.descripcion
  puts "Tarifa: $#{v.tarifa_por_hora}/hora"
  total += v.tarifa_por_hora
  puts "---"
end
puts "Si todos se rentan 1 hora: $#{total}"

```

Observa cómo el bloque `flota.each` trata a todos por igual: llama `encender` (heredado), `descripcion` (sobrescrito en cada hija con ayuda de `super`) y `tarifa_por_hora` (sobrescrito por completo). Ninguna línea pregunta "¿eres auto o moto?". Ese es el polimorfismo bien aplicado: el código cliente es simple porque el diseño es bueno.

Ejemplo 2: super en el constructor con validación extra

Un caso muy real: la hija quiere validar algo antes de dejar que el padre construya el objeto. Aquí la clase `CuentaAhorro` hereda de `CuentaBancaria` pero exige un depósito mínimo.

```

class CuentaBancaria
  attr_reader :titular, :saldo

  def initialize(titular, saldo_inicial)
    @titular = titular
    @saldo = saldo_inicial
  end

  def depositar(cantidad)
    @saldo += cantidad
    puts "Depósito de $#{cantidad}. Saldo: $#{@saldo}"
  end

  def retirar(cantidad)
    if cantidad <= @saldo
      @saldo -= cantidad
      puts "Retiro de $#{cantidad}. Saldo: $#{@saldo}"
    else
      puts "Fondos insuficientes"
    end
  end
end

class CuentaAhorro < CuentaBancaria
  DEPOSITO_MINIMO = 1000

  def initialize(titular, saldo_inicial)
    if saldo_inicial < DEPOSITO_MINIMO
      raise "El depósito mínimo es $#{DEPOSITO_MINIMO}"
    end
    super(titular, saldo_inicial)
    puts "Cuenta de ahorro creada para #{titular}"
  end

  def retirar(cantidad)
    puts "Aplicando comisión de $10 por retiro en cuenta de ahorro"
    super(cantidad + 10)
  end
end

cuenta = CuentaAhorro.new("Ana", 5000)
cuenta.depositar(1000) # heredado, funciona igual
cuenta.retirar(500)    # sobreescrito: avisa y suma comisión con super

```

Dos usos de `super` en un ejemplo: en ``initialize``, la hija valida primero y luego delega la construcción al padre; en ``retirar``, la hija agrega comportamiento (el aviso y la comisión) y luego deja que el padre haga el retiro real. Nota que ``super(cantidad + 10)`` pasa un argumento MODIFICADO: `super` no tiene que pasar lo mismo que recibió, puedes transformar los datos antes de delegar.

Ejemplo 3: Mixin con `include` para comportamiento compartido

Tres clases sin relación de herencia comparten la capacidad de "ser saludadas formalmente" mediante un módulo. Fíjate que `Cliente`, `Empleado` y `Proveedor` no tienen un padre común, pero el mixin les da el mismo comportamiento.

```

module Cortesia
  def saludar_formal
    puts "Buenos días, mi nombre es #{nombre}."
  end

  def despedirse
    puts "Fue un placer, #{nombre} se despide."
  end
end

class Cliente
  include Cortesia
  attr_reader :nombre

  def initialize(nombre)
    @nombre = nombre
  end
end

class Empleado
  include Cortesia
  attr_reader :nombre, :puesto

  def initialize(nombre, puesto)
    @nombre = nombre
    @puesto = puesto
  end
end

class Proveedor
  include Cortesia
  attr_reader :nombre, :empresa

  def initialize(nombre, empresa)
    @nombre = nombre
    @empresa = empresa
  end
end

[Cliente.new("Ana"), Empleado.new("Beto", "Cajero"), Proveedor.new("Carla",
"Lácteos SA")].each do |persona|
  persona.saludar_formal
end

```

Si mañana quieres agregar un método `agradecer` a la cortesía, lo agregas UNA vez en el módulo y las tres clases lo obtienen. Sin el módulo, tendrías que copiar el método en tres clases. Esa es la economía del mixin.

Ejemplo 4: include vs extend en el mismo programa

Este ejemplo pone frente a frente los dos mecanismos para que la diferencia te quede grabada. El módulo Auditoria se incluye en una clase (métodos de instancia) y se extiende en otra (métodos de clase).

```
module Auditoria
  def registrar(evento)
    puts "[#{Time.now.strftime('%H:%M:%S')}] #{evento}"
  end
end

class Pedido
  include Auditoria # los OBJETOS pedido registran eventos

  def initialize(folio)
    @folio = folio
  end

  def confirmar
    registrar("Pedido #{@folio} confirmado")
  end
end

class Sistema
  extend Auditoria # la CLASE sistema registra eventos globales

  def self.reiniciar
    registrar("Sistema reiniciado")
  end
end

pedido = Pedido.new("A-100")
pedido.confirmar # funciona: el objeto usa el método incluido

Sistema.reiniciar # funciona: la clase usa el método extendido

# pedido.registrar directo también funcionaría, es método de instancia
# Sistema.new.registrar # ESTO FALLARÍA: no es método de instancia
```

La regla de decisión en la práctica: ¿el comportamiento lo necesita cada objeto individual (cada pedido registra sus propios eventos)? ``include``. ¿Lo necesita la clase como un todo (el sistema registra eventos globales)? ``extend``.

Ejemplo 5: Comparable en una clase de productos

Con solo incluir Comparable y definir `<=>`, tu clase Productos gana comparaciones y ordenamiento. Este ejemplo además muestra ``sort`` y ``min`/`max`` funcionando gratis.


```

class Producto
  include Comparable
  attr_reader :nombre, :precio

  def initialize(nombre, precio)
    @nombre = nombre
    @precio = precio
  end

  def <=>(otro)
    @precio <=> otro.precio
  end

  def to_s
    "#{@nombre}: $#{@precio}"
  end
end

catalogo = [
  Producto.new("Laptop", 15000),
  Producto.new("Mouse", 300),
  Producto.new("Teclado", 800),
  Producto.new("Monitor", 4500)
]

puts "Ordenados por precio:"
puts catalogo.sort.map(&:to_s)

puts "\nEl más barato: #{catalogo.min}"
puts "El más caro: #{catalogo.max}"

barato = Producto.new("USB", 150)
caro = Producto.new("PC", 20000)
medio = Producto.new("Tablet", 6000)
puts "\n¿Tablet entre USB y PC? #{medio.between?(barato, caro)}"
puts "¿Mouse < Teclado? #{catalogo[1] < catalogo[2]}"

```

Sin Comparable, ordenar el catálogo habría requerido escribir tu propio algoritmo o un bloque de sort manual cada vez. Con el mixin, `sort`, `min`, `max` y `between?` salen gratis. Imagina este ahorro multiplicado por todas las clases comparables de un sistema real.

Ejemplo 6: Enumerable en una lista de tareas

Define `each` una vez, incluye Enumerable, y tu clase ListaTareas obtiene map, select, count, find y más. Este ejemplo simula una app de pendientes.

```
class ListaTareas
  include Enumerable

  def initialize
    @tareas = []
  end

  def agregar(descripcion)
    @tareas << { descripcion: descripcion, hecha: false }
  end

  def completar(indice)
    @tareas[indice][:hecha] = true
  end

  def each(&bloque)
    @tareas.each(&bloque)
  end
end

lista = ListaTareas.new
lista.agregar("Estudiar Ruby")
lista.agregar("Hacer ejercicio")
lista.agregar("Leer 20 páginas")
lista.completar(0)

puts "Total: #{lista.count}"
puts "Pendientes: #{lista.count { |t| !t[:hecha] }}"
puts "Hechas: #{lista.select { |t| t[:hecha] }.map { |t| t[:descripcion] }.inspect}"

primera_pendiente = lista.find { |t| !t[:hecha] }
puts "Primera pendiente: #{primera_pendiente[:descripcion]}"

puts "\nTodas las tareas:"
lista.each_with_index do |t, i|
  marca = t[:hecha] ? "[x]" : "[ ]"
  puts "#{i + 1}. #{marca} #{t[:descripcion]}"
end
```

Todo lo que ves (count con bloque, select, find, each_with_index) viene de Enumerable. Tu clase solo aportó `each` y la lógica de negocio (agregar, completar). Este patrón aparece en todas partes: barajas de cartas, carritos de compra, playlists, inventarios.

Ejemplo 7: Namespace para un mini sistema de tienda

Dos áreas del sistema necesitan una clase Utilidad con el mismo nombre pero distinto contenido. El namespace resuelve el choque y organiza el código por temas.

```
module Tienda
  module Caja
    class Utilidad
      def self.calcular(ventas, costos)
        ventas - costos
      end
    end

    class Ticket
      def initialize(total)
        @total = total
      end

      def imprimir
        puts "=== TICKET ==="
        puts "Total: ${@total}"
        puts "Gracias por su compra"
      end
    end
  end

  module Almacen
    class Utilidad
      def self.calcular(existencias, minimo)
        existencias < minimo ? "PEDIR MÁS" : "OK"
      end
    end
  end
end

puts Tienda::Caja::Utilidad.calcular(10000, 6000)
puts Tienda::Almacen::Utilidad.calcular(5, 10)

Tienda::Caja::Ticket.new(250).imprimir
```

Fíjate que hay namespaces ANIDADOS: `Tienda::Caja::Utilidad`. Puedes anidar módulos tanto como necesites para reflejar la estructura de tu proyecto. En proyectos reales, cada módulo suele vivir en su propia carpeta de archivos, y el namespace refleja esa organización.

Ejemplo 8: Procesador de texto con strings avanzados

Un programa completo que usa interpolación, `gsub` con regex, `strip`, `split`, `upcase` y heredoc para generar un reporte limpio a partir de datos "sucios". Esto es muy parecido a lo que harías en un trabajo real limpiando datos de un formulario.

```

# Datos "sucios" como los que llegan de un formulario real
registros_sucios = [
  " ANA LÓPEZ , ana.lopez@correo.com , 555-1234 ",
  "beto ramírez, BETO@CORREO.COM,555 9876",
  " Carla Núñez,,,carla.nunez@correo.com, 555-0000 "
]

def limpiar_registro(linea)
  # 1. Dividir por comas y quitar vacíos
  campos = linea.split(",").map(&:strip).reject(&:empty?)

  # 2. El primer campo es el nombre: capitalizar cada palabra
  nombre = campos[0].split.map(&:capitalize).join(" ")

  # 3. Buscar el campo que parece email (contiene @)
  email = campos.find { |c| c.include?("@") }.downcase

  # 4. Buscar el campo con dígitos: normalizar el teléfono
  telefono_crudo = campos.find { |c| c.match?(/\d/) }
  telefono = telefono_crudo.gsub(/\D/, "") # quitar todo lo que no sea dígito

  { nombre: nombre, email: email, telefono: telefono }
end

registros = registros_sucios.map { |r| limpiar_registro(r) }

# 5. Generar el reporte con heredoc
reporte = <<~REPORTE
REPORTE DE CONTACTOS
=====
Total de registros: #{registros.length}

  #{registros.map { |r| "- #{r[:nombre]} | #{r[:email]} | Tel: #{r[:telefono]}"
}.join("\n")}

  Generado el #{Time.now.strftime('%d/%m/%Y a las %H:%M')}
REPORTE

puts reporte

```

Recorre el programa con calma: ``split`` + ``strip`` + ``reject`` limpian los campos; ``map(&:capitalize)`` arregla el nombre; ``find`` con ``include?("@")`` localiza el email sin importar su posición; ``gsub(/\D/, "")`` deja solo dígitos del teléfono (\D es "todo lo que NO sea dígito"); y el heredoc arma el reporte final con interpolación. Cada técnica de strings de la unidad aparece en un contexto real.

Ejemplo 9: Validador de contraseñas con regex y métodos de string

Un validador que combina varios métodos de string y una pizca de regex para revisar reglas de contraseña. Útil y directo.

```
def validar_password(password)
  errores = []

  errores << "mínimo 8 caracteres" if password.length < 8
  errores << "debe tener al menos una mayúscula" unless password.match?(/[A-Z]/)
  errores << "debe tener al menos una minúscula" unless password.match?(/[a-z]/)
  errores << "debe tener al menos un número" unless password.match?(/\d/)
  errores << "no debe tener espacios" if password.include?(" ")

  if errores.empty?
    puts "Password '#{password}' válido"
    true
  else
    puts "Password '#{password}' inválido:"
    errores.each { |e| puts "  - #{e}" }
    false
  end
end

validar_password("abc")
validar_password("Password123")
validar_password("pass word1")
```

El método ``match?`` devuelve true/false según si el string cumple el patrón, sin el costo de crear objetos de captura: es la forma moderna y rápida de preguntar "¿cumple el patrón?". Úsalo en validaciones.

Ejemplo 10: Heredoc como plantilla de email reutilizable

Una pequeña función que genera emails personalizados con heredoc e interpolación. El tipo de código que escribirías para notificaciones automáticas.

```

def email_bienvenida(nombre, curso)
  <<~EMAIL
    Hola #{nombre.split.map(&:capitalize).join(" ")},

    ¡Bienvenido a Lucas EGA Academy!

    Te inscribiste al curso "#{curso}".
    Puedes empezar hoy mismo en lucasegaacademy.com.

    Consejos para aprovecharlo:
    - Dedicar 30 minutos diarios
    - Escribe el código tú mismo, no solo lo leas
    - Repite los ejercicios hasta que salgan sin mirar

    ¡Éxito en tu aprendizaje!
    El equipo de Lucas EGA Academy
  EMAIL
end

puts email_bienvenida("ana lópez", "Curso Completo de Ruby")
puts "---"
puts email_bienvenida("BETTO RUIZ", "Curso de HTML5")

```

Observa que el nombre se normaliza dentro de la interpolación: ``nombre.split.map(&:capitalize).join(" ")`` convierte "ana lópez" o "BETTO RUIZ" en formato presentable. La plantilla queda legible como documento, no como código enredado.

Práctica

Ejercicio 1: Jerarquía de figuras geométricas

Crea una clase `Figura` con un método ``area`` que devuelva 0 y un método ``descripcion`` que devuelva "Figura genérica". Luego crea `Cuadrado` (lado) y `Circulo` (radio) que hereden de `Figura`, sobrescriban ``area`` con la fórmula real y extiendan ``descripcion`` con `super`. Usa `Math::PI` para el círculo.

Solución explicada:

```

class Figura
  def area
    0
  end

  def descripcion
    "Figura genérica"
  end
end

class Cuadrado < Figura
  def initialize(lado)
    @lado = lado
  end

  def area
    @lado * @lado
  end

  def descripcion
    super + " (cuadrado de lado #{@lado})"
  end
end

class Circulo < Figura
  def initialize(radio)
    @radio = radio
  end

  def area
    Math::PI * @radio * @radio
  end

  def descripcion
    super + " (círculo de radio #{@radio})"
  end
end

figuras = [Cuadrado.new(4), Circulo.new(3), Figura.new]
figuras.each do |f|
  puts "#{f.descripcion} - área: #{f.area.round(2)}"
end

```


La clase Figura define el "contrato": toda figura sabe calcular su área y describirse. Las hijas sobrescriben `area` por completo (la fórmula no tiene nada que ver con el 0 del padre) pero extienden `descripcion` con super porque sí aprovechan el texto base. El `each` final demuestra polimorfismo: trata a las tres por igual.

Ejercicio 2: Empleados con super en initialize

Crea una clase Empleado con nombre y sueldo_base, y un método `sueldo\_total` que devuelva el sueldo base. Crea Gerente que herede de Empleado, reciba además un bono, y sobrescriba `sueldo\_total` usando super para sumar el bono.

Solución explicada:

```
class Empleado
  attr_reader :nombre

  def initialize(nombre, sueldo_base)
    @nombre = nombre
    @sueldo_base = sueldo_base
  end

  def sueldo_total
    @sueldo_base
  end
end

class Gerente < Empleado
  def initialize(nombre, sueldo_base, bono)
    super(nombre, sueldo_base)
    @bono = bono
  end

  def sueldo_total
    super + @bono
  end
end

ana = Empleado.new("Ana", 10000)
beto = Gerente.new("Beto", 12000, 5000)
puts "#{ana.nombre}: ${ana.sueldo_total}"
puts "#{beto.nombre}: ${beto.sueldo_total}"
```

El patrón se repite: `initialize` delega al padre con super y agrega lo propio; `suelo\_total` reutiliza el cálculo del padre y le suma el bono. Si mañana el cálculo base cambia (por ejemplo, descuentos), solo se toca el padre y el gerente lo hereda automáticamente.

Ejercicio 3: Módulo Descontable como mixin

Crea un módulo Descontable con un método `aplicar\_descuento(porcentaje)` que devuelva el precio con descuento. Inclúyelo en las clases Libro y Curso (no relacionadas por herencia), cada una con su atributo precio. Prueba que ambas puedan aplicar descuentos.

Solución explicada:

```
module Descontable
  def aplicar_descuento(porcentaje)
    @precio * (1 - porcentaje / 100.0)
  end
end

class Libro
  include Descontable
  attr_reader :precio, :titulo

  def initialize(titulo, precio)
    @titulo = titulo
    @precio = precio
  end
end

class Curso
  include Descontable
  attr_reader :precio, :nombre

  def initialize(nombre, precio)
    @nombre = nombre
    @precio = precio
  end
end

libro = Libro.new("Ruby básico", 300)
curso = Curso.new("Ruby completo", 1500)
puts "Libro con 10% off: ${libro.aplicar_descuento(10)}"
puts "Curso con 20% off: ${curso.aplicar_descuento(20)}"
```

Libro y Curso no tienen padre común, así que la herencia sería forzada; el módulo es la herramienta correcta. Nota que el módulo usa `@precio` directamente: funciona porque cuando el método se ejecuta, `self` es el objeto que lo incluyó y ese objeto sí tiene `@precio`. Es un acuerdo implícito: documenta qué atributos espera tu módulo.

Ejercicio 4: include vs extend, demuéstalo

Crea un módulo Contador con un método `incrementar` que sume 1 a un contador. Crea una clase Visita que lo incluya y una clase Estadísticas que lo extienda. Demuestra con código qué funciona y qué falla en cada caso (usa `begin/rescue` para atrapar el error y mostrarlo amablemente).

Solución explicada:

```

module Contador
  def incrementar
    @contador ||= 0
    @contador += 1
    puts "Contador: #{@contador}"
  end
end

class Visita
  include Contador
end

class Estadisticas
  extend Contador
end

# include: funciona en el OBJETO
Visita.new.incrementar
Visita.new.incrementar

# extend: funciona en la CLASE
Estadisticas.incrementar

# Lo contrario falla: lo atrapamos para ver el error
begin
  Visita.incrementar
rescue NoMethodError => e
  puts "Error esperado: #{e.message[0..60]}..."
end

begin
  Estadisticas.new.incrementar
rescue NoMethodError => e
  puts "Error esperado: #{e.message[0..60]}..."
end

```

Este ejercicio te obliga a sentir la diferencia en lugar de solo leerla: `include` pone el método en las instancias, `extend` en la clase. Los dos rescates muestran exactamente el error que verías en un programa real si los confundes.

Ejercicio 5: Comparable para ordenar películas por calificación

Crea una clase Pelicula con titulo y calificacion (0-10). Incluye Comparable definiendo <=> por calificación. Crea 4 películas, ordénalas, muestra la mejor y la peor, y pregunta si una película está entre otras dos con between?.

Solución explicada:

```
class Pelicula
  include Comparable
  attr_reader :titulo, :calificacion

  def initialize(titulo, calificacion)
    @titulo = titulo
    @calificacion = calificacion
  end

  def <=>(otra)
    @calificacion <=> otra.calificacion
  end

  def to_s
    "#{@titulo} (#{@calificacion}/10)"
  end
end

pelis = [
  Pelicula.new("El Padrino", 9.2),
  Pelicula.new("Titanic", 7.8),
  Pelicula.new("Interestelar", 8.7),
  Pelicula.new("Shrek", 8.1)
]

puts "De peor a mejor:"
pelis.sort.each { |p| puts "  #{p}" }
puts "La mejor: #{pelis.max}"
puts "La peor: #{pelis.min}"

shrek = pelis.find { |p| p.titulo == "Shrek" }
titanic = pelis.find { |p| p.titulo == "Titanic" }
inter = pelis.find { |p| p.titulo == "Interestelar" }
puts "¿Shrek entre Titanic e Interestelar? #{shrek.between?(titanic, inter)}"
```

Con un solo método `<=>` obtuviste `sort`, `min`, `max` y `between?`. Fíjate que `between?` necesita objetos del mismo tipo en los extremos: compara calificaciones, no títulos. Si mezclaras tipos, `<=>` devolvería `nil` y `between?` fallaría: Comparable exige coherencia.

Ejercicio 6: Enumerable para una playlist

Crea una clase `Playlist` que guarde canciones (título y duración en segundos). Incluye `Enumerable` definiendo `each`. Luego usa al menos 4 métodos de `Enumerable`: duración total con `sum`, canciones largas con `select`, buscar una por título con `find`, y verificar si todas duran menos de 5 minutos con `all?`.

Solución explicada:

```
class Playlist
  include Enumerable

  def initialize(nombre)
    @nombre = nombre
    @canciones = []
  end

  def agregar(titulo, duracion_seg)
    @canciones << { titulo: titulo, duracion: duracion_seg }
  end

  def each(&bloque)
    @canciones.each(&bloque)
  end
end

lista = Playlist.new("Rock clásico")
lista.agregar("Bohemian Rhapsody", 354)
lista.agregar("Smells Like Teen Spirit", 301)
lista.agregar("Stairway to Heaven", 482)
lista.agregar("Creep", 238)

puts "Duración total: #{lista.sum { |c| c[:duracion] }} segundos"
largas = lista.select { |c| c[:duracion] > 300 }
puts "Canciones de más de 5 min: #{largas.map { |c| c[:titulo] }.inspect}"
encontrada = lista.find { |c| c[:titulo].include?("Creep") }
puts "Encontrada: #{encontrada[:titulo]}"
puts "¿Todas duran menos de 10 min? #{lista.all? { |c| c[:duracion] < 600 }}"
puts "¿Alguna dura más de 7 min? #{lista.any? { |c| c[:duracion] > 420 }}"
```

Tu clase aportó `each` y la lógica de agregar; `Enumerable` aportó `sum`, `select`, `find`, `all?` y `any?`. Este es exactamente el patrón que usan `Array` y `Hash` por dentro. Cuando diseñes colecciones propias, este ejercicio es tu plantilla.

Ejercicio 7: Limpieza de texto con `gsub` y métodos de `string`

Escribe un método `normalizar_nombre` que reciba un `string` sucio (mayúsculas mezcladas, espacios de más, números colados) y devuelva el nombre limpio: solo letras y espacios simples, cada palabra capitalizada. Ejemplo: " aNA---123 LÓPEZ__ " debe quedar "Ana López".

Solución explicada:

```
def normalizar_nombre(sucio)
  # 1. Quitar todo lo que no sea letra (con acentos/ñ) ni espacio
  solo_letras = sucio.gsub(/[^a-zA-ZáéíóúÁÉÍÓÚñÑü\ ]/, " ")
  # 2. Colapsar espacios múltiples y quitar extremos
  compacto = solo_letras.gsub(/\s+/, " ").strip
  # 3. Capitalizar cada palabra
  compacto.split.map(&:capitalize).join(" ")
end

puts normalizar_nombre(" aNA---123 LÓPEZ__ ").inspect # "Ana López"
puts normalizar_nombre("jOSÉ marÍA pérez99").inspect # "José María Pérez"
puts normalizar_nombre(" CARLOS ").inspect           # "Carlos"
```

Tres pasos, tres técnicas: la `regex` `[^...]` filtra caracteres, `\s+` colapsa espacios, y `split/map/join` capitaliza. Fíjate en el orden: primero filtras, luego compactas, luego capitalizas. Si capitalizaras antes de quitar los números, desperdiciarías trabajo. El orden de las operaciones en limpieza de texto sí importa.

Ejercicio 8: Reporte con heredoc y namespaces

Crea un módulo `Escuela` con dos clases: `Alumno` (`nombre`, `promedio`) y `Reporte`. El método `Reporte.generar` debe recibir un arreglo de alumnos y devolver (con heredoc) un reporte multilínea que liste a cada alumno con su promedio y su estatus (Aprobado si `promedio >= 6`, Reprobado si no), más el promedio general del grupo. Usa interpolación en todo.

Solución explicada:

```

module Escuela
  class Alumno
    attr_reader :nombre, :promedio

    def initialize(nombre, promedio)
      @nombre = nombre
      @promedio = promedio
    end

    def estatus
      @promedio >= 6 ? "Aprobado" : "Reprobado"
    end
  end

  class Reporte
    def self.generar(alumnos)
      promedio_general = alumnos.sum(&:promedio) / alumnos.length.to_f
      lineas = alumnos.map do |a|
        "- #{a.nombre}: #{a.promedio} (#{a.estatus})"
      end.join("\n")

      <<~REPORTE
      REPORTE DE CALIFICACIONES
      =====
      #{lineas}

      Promedio general del grupo: #{promedio_general.round(2)}
      Total de alumnos: #{alumnos.length}
      Aprobados: #{alumnos.count { |a| a.estatus == "Aprobado" }}
      REPORTE
    end
  end
end

grupo = [
  Escuela::Alumno.new("Ana", 9.2),
  Escuela::Alumno.new("Beto", 5.5),
  Escuela::Alumno.new("Carla", 7.8)
]
puts Escuela::Reporte.generar(grupo)

```


Junta todo lo de la unidad: namespace (Escuela::Alumno), interpolación, heredoc, métodos de Enumerable (sum, count, map). El método generar es de clase (self.generar) porque no necesita estado: recibe datos y devuelve texto. Nota el detalle ``alumnos.length.to_f``: sin el `to_f`, la división de enteros truncaría el promedio.

Resumen de la unidad

- La herencia (class Hija < Padre) le da a la hija todos los métodos del padre; úsala cuando se cumpla la prueba del "es un" (un Auto es un Vehículo).
- ``super`` llama al método del mismo nombre en el padre: ``super`` solo pasa los argumentos automáticamente, ``super()`` no pasa ninguno, ``super(a, b)`` pasa los que indiques.
- Sobreescibir es redefinir un método del padre en la hija; combínalo con `super` cuando quieras extender en lugar de reemplazar, y respeta el contrato del método original.
- Mantén las jerarquías cortas (2-3 niveles); para "partes de" usa composición, no herencia.
- Un módulo es un paquete de métodos que no se puede instanciar; sirve para mixins (compartir comportamiento) y para namespaces (organizar nombres).
- ``include`` agrega métodos de instancia (los usan los objetos); ``extend`` agrega métodos de clase (los usa la clase). Casi siempre querrás `include`.
- Comparable te da `<`, `>`, `<=`, `>=`, `==`, `sort`, `min`, `max` y `between?` definiendo solo `<=`; Enumerable te da `map`, `select`, `find`, `count`, `sum` y más definiendo solo `each`.
- Los namespaces (Modulo::Clase) evitan choques de nombres y organizan proyectos grandes; el operador `::` arma el nombre completo.
- La interpolación `#{}` dentro de comillas dobles (o heredoc) es la forma más legible de construir texto dinámico; llama a `to_s` automáticamente.
- Métodos de string esenciales: `upcase/downcase/capitalize`, `strip`, `length`, `empty?`, `include?`, `start_with?/end_with?`, `split/join`, `sub/gsub` (con o sin `!` para modificar el original).
- `gsub` con regex resuelve limpieza y reemplazos complejos: `\d`, `\w`, `\s`, `[^...]` para negar, paréntesis para capturar y `\1`, `\2` para reutilizar; prueba tus regex antes de usarlas en serio.
- El heredoc (`<<~MARCA`) escribe textos multilínea legibles como documentos; la variante `<<~'MARCA'` no interpola.
- ``freeze`` congela un string para que nadie lo modifique por accidente; congela tus constantes de texto.

- Trampas clásicas: gets trae "\n" (usa chomp/strip), upcase sin ! no modifica el original, comillas simples no interpolan, y "Edad: " + 25 truena (usa interpolación).

UNIDAD 8: Errores, archivos y tiempo

Felicidades por llegar a la Unidad 8. Hasta ahora tus programas vivían en un mundo perfecto: los datos siempre eran correctos, los archivos siempre existían y el tiempo no importaba. Pero los programas de verdad viven en el mundo real, donde el usuario escribe letras cuando le pides números, donde el archivo que buscas no está, donde la red se cae a la mitad de una operación y donde necesitas saber qué hora es para registrar cuándo pasó cada cosa.

Esta unidad te da las herramientas para programar como adulto: manejar errores sin que tu programa se muera, leer y escribir archivos para guardar información que sobreviva cuando el programa termina, recorrer directorios para trabajar con muchos archivos a la vez, y manejar fechas y horas para registrar eventos, calcular duraciones y medir qué tan rápido corre tu código.

Estos tres temas comparten una idea: tu programa deja de ser una cajita aislada y empieza a interactuar con el mundo exterior. Los errores son el mundo diciéndote "algo salió mal"; los archivos son tu memoria a largo plazo; el tiempo es el contexto en el que todo ocurre. Dominar esta unidad es lo que separa un script de juguete de una aplicación de verdad.

Al terminar vas a escribir programas que no se caen ante datos malos (rescatan el error, avisan y siguen), que guardan y leen información en disco (bitácoras, configuraciones, reportes), que procesan carpetas enteras de archivos automáticamente, y que trabajan con fechas como un profesional: formatearlas, compararlas, sumar días y medir rendimiento. Vamos a ensuciarnos las manos con el mundo real.

Explicación detallada

Qué son las excepciones y por qué no debes temerles

En Ruby, cuando algo sale mal, el programa no se queda callado: lanza una excepción. Una excepción es un objeto que representa un error, y "lanzar" significa interrumpir el flujo normal del programa para decir "aquí pasó algo malo". Si nadie la atrapa, el programa termina y Ruby te muestra el error en la consola con una pila de llamadas (stack trace) que indica dónde ocurrió.

Mira lo más simple:

```
puts 10 / 0  
# ZeroDivisionError: divided by 0
```

Dividir entre cero lanza una excepción `ZeroDivisionError` y el programa muere ahí. Lo mismo pasa si abres un archivo que no existe (`Errno::ENOENT`), si conviertes mal un dato, si accedes a una posición inválida o si llamas un método que no existe (`NoMethodError`). Cada tipo de error tiene su propia clase de excepción, y eso es una ventaja: puedes reaccionar diferente según qué salió mal.

La analogía: una excepción es como la alarma de incendios de un edificio. Cuando suena, todo el mundo deja lo que está haciendo y sigue el protocolo de evacuación. En tu programa, el "protocolo" lo defines tú con `begin/rescue`: en lugar de que el edificio (tu programa) se queme entero (se cierre), atrapas la alarma, la atiendes y decides qué hacer: reintentar, usar un valor por defecto, avisar al usuario o, si es grave, dejar que el programa termine pero de forma ordenada.

El error mental más común de los principiantes es ver las excepciones como enemigas. No lo son: son información. Un programa que truena con un mensaje claro es mejor que uno que sigue corriendo con datos corruptos en silencio. La meta no es "que nunca haya errores", sino "que cuando haya errores, el programa se comporte con dignidad".

Otro error típico, en el extremo opuesto, es atrapar TODAS las excepciones a ciegas para que "nunca truene". Eso es como desconectar la alarma de incendios porque te molesta el ruido: el fuego sigue ahí, solo que ya no te enteras. Más adelante verás por qué rescatar `Exception` (la madre de todas) es casi siempre una mala idea.

`begin/rescue/ensure/else`: el protocolo completo

La estructura para manejar excepciones tiene cuatro partes. Piensa en ellas como el protocolo de emergencias de un restaurante:

```
begin
  # Código que puede fallar (la cocina)
  numero = Integer("abc") # esto lanza ArgumentError
  puts "El número es #{numero}"
rescue ArgumentError => e
  # Qué hacer si falla (el plan B)
  puts "No escribiste un número válido: #{e.message}"
else
  # Qué hacer si TODO salió bien (el brindis)
  puts "Conversión exitosa"
ensure
  # Qué hacer SIEMPRE, falle o no (apagar las luces al salir)
  puts "Fin del intento"
end
```

Vamos parte por parte:

- **begin:** marca el inicio de la zona peligrosa, el código que puede lanzar una excepción. Todo lo que esté entre begin y rescue está "protegido".
- **rescue:** se ejecuta SOLO si ocurrió una excepción en el begin. Aquí pones tu plan B: mensaje amable, valor por defecto, reintento, registro en bitácora. Puedes tener varios rescue para distintos tipos de error (lo veremos en la siguiente sección).
- **else:** se ejecuta SOLO si NO hubo excepción. Es el lugar para el código que depende de que todo haya salido bien. Mucha gente no lo conoce y mete ese código al final del begin, pero separarlo hace tu intención más clara: "esto solo corre si no hubo error".
- **ensure:** se ejecuta SIEMPRE, haya habido excepción o no, incluso si el rescue también falló o si hubo un return en medio. Es para limpieza obligatoria: cerrar archivos, cerrar conexiones, liberar recursos.

La variable `=> e` en el rescue guarda el objeto de la excepción. Con `e.message` obtienes la descripción del error y con `e.class` su tipo. En depuración también sirve `e.backtrace`, que muestra la pila de llamadas (las primeras líneas te dicen dónde empezó el problema).

Un detalle elegante de Ruby: en un método, puedes omitir el begin si todo el cuerpo del método es la zona protegida:

```
def convertir(texto)
  numero = Integer(texto)
  puts "Convertido: #{numero}"
rescue ArgumentError => e
  puts "Error: #{e.message}"
end
```

Ruby entiende que el `rescue` al final de un método protege todo el método. Es azúcar sintáctico: menos indentación, misma protección. Úsalo cuando TODO el método puede fallar; usa `begin/rescue` explícito cuando solo una parte es riesgosa.

Error común: poner código que NO puede fallar dentro del `begin` "por si acaso". Eso es como ponerle casco a una silla: no protege nada y confunde. El `begin` debe contener solo el código riesgoso; lo demás va antes o en el `else`.

Rescatar errores específicos (y por qué no rescatar `Exception`)

Puedes (y debes) tener varios bloques `rescue`, uno por cada tipo de error que sepas manejar. Ruby los prueba en orden, de arriba hacia abajo, y ejecuta el primero que coincida:

```
def dividir(a, b)
  resultado = a / b
  puts "Resultado: #{resultado}"
rescue ZeroDivisionError
  puts "No puedes dividir entre cero"
rescue TypeError
  puts "Solo puedo dividir números, no #{b.class}"
end

dividir(10, 2)    # Resultado: 5
dividir(10, 0)    # No puedes dividir entre cero
dividir(10, "x")  # Solo puedo dividir números, no String
```

El orden importa: pon primero las excepciones MÁS específicas y al final las más generales, porque Ruby usa la primera que coincida. Si pones `StandardError` primero, los `rescue` específicos de abajo nunca se ejecutarán (Ruby incluso te avisa con una advertencia en algunos casos, pero no siempre).

La jerarquía de excepciones que debes conocer:

- **Exception:** la madre de todas. NO la rescates (casi) nunca.

- **StandardError**: la madre de los errores "normales" de programa. Casi todos los errores que te interesan cuelgan de aquí: `ArgumentError`, `TypeError`, `ZeroDivisionError`, `NoMethodError`, `RuntimeError`, `IOError`, etc.
- **Errores especiales** que cuelgan directo de `Exception` y NO de `StandardError`: `SystemExit` (cuando el programa termina con `exit`), `NoMemoryError`, `SignalException` (Ctrl+C). Estos NO se atrapan con un ``rescue`` a secas, y está bien que así sea: no quieres atrapar la señal de "el usuario quiere salir".

Cuando escribes ``rescue`` sin especificar clase, Ruby rescata `StandardError` y sus hijas. Eso es lo correcto el 95% de las veces. El antipatrón mortal es ``rescue Exception``:

Código malo (atrapa hasta la salida del programa):

```
begin
  # ... código ...
rescue Exception => e # MAL: atrapa SystemExit, NoMemoryError, Ctrl+C
  puts "Algo pasó"
end
```

¿Por qué es malo? Porque si el usuario presiona Ctrl+C para detener tu programa, tu `rescue` lo atrapa y el programa NO se detiene. Porque si Ruby se queda sin memoria, tu programa finge que todo está bien. ``rescue Exception`` es la manguera que apaga la alarma de incendios: esconde el problema en lugar de resolverlo.

Código bueno (específico y honesto):

```
begin
  # ... código ...
rescue ZeroDivisionError
  puts "División entre cero"
rescue StandardError => e
  puts "Error inesperado: #{e.class}: #{e.message}"
end
```

Si de verdad necesitas un "atrapa todo" como última red de seguridad (por ejemplo, en el loop principal de un servidor que no debe caerse), usa ``rescue StandardError``, documenta por qué, y REGISTRA el error completo (clase, mensaje y backtrace) en lugar de tragarlo en silencio. Un error silenciado es un bug que vas a perseguir durante semanas.

raise: lanzar tus propios errores

Hasta ahora las excepciones las lanzaba Ruby. Pero tú también puedes (y debes) lanzar las tuyas cuando detectes una situación inválida en tu lógica de negocio. Para eso sirve `raise`: crea una excepción y la lanza inmediatamente.

```
def retirar(saldo, cantidad)
  raise "La cantidad debe ser positiva" if cantidad <= 0
  raise "Fondos insuficientes: tienes #{saldo}" if cantidad > saldo
  saldo - cantidad
end

begin
  puts retirar(1000, 500)    # 500
  puts retirar(1000, 1500)  # lanza RuntimeError
rescue RuntimeError => e
  puts "No se pudo retirar: #{e.message}"
end
```

`raise "mensaje"` crea una excepción de tipo `RuntimeError` con ese mensaje. Es la forma rápida de validar. También puedes lanzar un tipo específico: `raise ArgumentError, "mensaje"` o `raise MiErrorPersonalizado, "mensaje"` (veremos clases propias en la siguiente sección).

Piensa en `raise` como el guardia de seguridad de tus métodos: su trabajo es impedir que entren datos inválidos. Sin validaciones con `raise`, los datos malos se cuelan hasta lo profundo de tu programa y el error aparece lejos de donde se originó, haciendo la depuración un infierno. Con `raise` al inicio del método (esto se llama "programación defensiva" o "fallar rápido"), el error aparece justo donde está el problema, con un mensaje que TÚ escribiste.

Buena práctica: los mensajes de `raise` deben decir QUÉ está mal y, si es posible, QUÉ se esperaba:

Código malo (mensaje críptico):

```
raise "Error" if edad < 0
```

Código bueno (mensaje útil):

```
raise ArgumentError, "La edad no puede ser negativa (recibido: #{edad})" if edad < 0
```


Tu yo del futuro, depurando a las 2 de la mañana, te va a agradecer cada mensaje claro que escribas hoy. Incluye los valores recibidos: ver "recibido: -5" te dice todo al instante.

Otro uso legítimo de `raise` es relanzar una excepción después de hacer algo con ella. Dentro de un `rescue`, un `raise` a secas relanza la misma excepción que atrapaste, para que alguien más arriba en la cadena también pueda manejarla:

```
begin
  # operación riesgosa
rescue IOError => e
  puts "Registrando en bitácora: #{e.message}"
  raise # la relanzo: ya la registré, pero no la puedo resolver aquí
end
```

Esto es común en programas por capas: la capa baja registra el detalle técnico, la capa alta decide qué mostrarle al usuario.

Crear tus propias clases de error

Cuando tu programa crece, `raise "mensaje"` se queda corto: todos tus errores son `RuntimeError` y quien los atrape no puede distinguir "fondos insuficientes" de "cuenta bloqueada". La solución es crear tus propias clases de excepción, que es ridículamente fácil: heredan de `StandardError` (o de una clase más específica).

```

class FondosInsuficientes < StandardError; end
class CuentaBloqueada < StandardError; end
class MontoInvalido < StandardError; end

def retirar(saldo, cantidad, bloqueada)
  raise CuentaBloqueada, "La cuenta está bloqueada" if bloqueada
  raise MontoInvalido, "El monto debe ser positivo (recibido: #{cantidad})" if
cantidad <= 0
  raise FondosInsuficientes, "Tienes #{saldo}, quieres #{cantidad}" if cantidad >
saldo
  saldo - cantidad
end

begin
  retirar(1000, 1500, false)
rescue FondosInsuficientes => e
  puts "Sin dinero: #{e.message}"
rescue CuentaBloqueada => e
  puts "Habla al banco: #{e.message}"
rescue MontoInvalido => e
  puts "Revisa la cantidad: #{e.message}"
end

```

Cada error tiene su clase, su rescue y su manejo distinto. El código cliente puede decidir: "si es fondos insuficientes, ofrezco un préstamo; si es cuenta bloqueada, muestro el teléfono del banco". Con RuntimeError genérico, todo eso sería imposible sin andar analizando el texto del mensaje (frágil y feo).

Puedes organizar tus errores en jerarquías, igual que cualquier clase. Un truco elegante: crea una clase base para todos los errores de tu aplicación y haz que los específicos hereden de ella. Así quien quiera puede atrapar "cualquier error de mi app" con un solo rescue:

```

class ErrorBancario < StandardError; end
class FondosInsuficientes < ErrorBancario; end
class CuentaBloqueada < ErrorBancario; end

begin
  # ... operaciones bancarias ...
rescue ErrorBancario => e
  puts "Error del banco: #{e.message}"
  # atrapa FondosInsuficientes Y CuentaBloqueada, porque son hijas
end

```

Buena práctica: define tus clases de error en un solo archivo (errores.rb) al inicio del proyecto, con nombres descriptivos que terminen en algo claro (Error, Exception o nada, pero sé consistente). No abuses: si un error solo ocurre en un lugar y nadie lo va a rescatar de forma especial, un ArgumentError con buen mensaje basta. Crea clases propias para los errores que tu programa necesita DISTINGUIR al atraparlos.

Archivos: por qué tu programa necesita memoria a largo plazo

Todo lo que tu programa guarda en variables desaparece cuando termina. Las variables son como la memoria de corto plazo: útiles mientras trabajas, inútiles cuando apagas. Los archivos son la memoria a largo plazo: ahí guardas bitácoras, configuraciones, datos de usuarios, reportes y todo lo que debe sobrevivir al cierre del programa.

Ruby hace trabajar con archivos algo casi trivial. La clase File tiene métodos para los casos simples y la forma `File.open` con bloque para los casos controlados. Vamos de lo simple a lo completo.

Lectura simple: File.read

Si solo quieres el contenido completo de un archivo como un string, una línea basta:

```
contenido = File.read("notas.txt")
puts contenido
```

`File.read` abre el archivo, lee TODO de un golpe y lo cierra. Perfecto para archivos pequeños y medianos (configuraciones, plantillas, archivos de texto normales). No lo uses con archivos gigantescos (un log de 2 GB) porque cargaría todo en memoria: para esos casos lee por líneas, como veremos.

Escritura simple: File.write

Igual de directo para guardar un string en un archivo:

```
File.write("salida.txt", "Hola mundo\nSegunda línea\n")
```

Si el archivo no existe, lo crea. Si existe, LO SOBREEESCRIBE por completo (cuidado: el contenido anterior se pierde). Para agregar sin borrar, necesitas el modo "a" (append), que veremos con `File.open`.

File.open con bloque: la forma segura

Cuando necesitas más control (leer línea por línea, escribir varias cosas, elegir el modo), usa `File.open` con un bloque. La magia del bloque es que Ruby CIERRA el archivo automáticamente al terminar, incluso si ocurre una excepción en medio:

```
File.open("bitacora.txt", "a") do |archivo|
  archivo.puts "#{Time.now}: El programa inició"
  archivo.puts "#{Time.now}: Proceso completado"
end
# aquí el archivo YA está cerrado, garantizado
```

Compara con la forma manual, donde TÚ eres responsable de cerrar:

Código malo (se te puede olvidar cerrar, o una excepción lo impide):

```
archivo = File.open("bitacora.txt", "a")
archivo.puts "algo"
# si aquí truena algo, el archivo queda abierto
archivo.close # si se te olvida esta línea, hay fuga de recursos
```

Código bueno (el bloque cierra solo):

```
File.open("bitacora.txt", "a") do |archivo|
  archivo.puts "algo"
end # cerrado automáticamente, pase lo que pase
```

Un archivo abierto que nunca se cierra es una fuga de recursos: el sistema operativo tiene un límite de archivos abiertos, y además los datos en el buffer pueden no guardarse en disco. La regla es simple: SIEMPRE usa `File.open` con bloque, a menos que tengas una razón muy específica para no hacerlo.

Modos de apertura: la letra que lo cambia todo

El segundo argumento de `File.open` es el modo, una letrita que define qué puedes hacer con el archivo y qué pasa con su contenido. Los que vas a usar el 99% del tiempo:

- **"r"** (read, es el modo por defecto): solo leer. Falla si el archivo no existe.
- **"w"** (write): escribir. CREA el archivo si no existe; si existe, BORRA todo su contenido. Peligroso si lo confundes.
- **"a"** (append): agregar al final. Crea si no existe; si existe, conserva el contenido y escribe al final. El modo de las bitácoras.

- **"r+":** leer y escribir sin borrar.
- **"w+":** leer y escribir, borrando el contenido.

```
# Leer línea por línea (ideal para archivos grandes)
File.open("datos.txt", "r") do |f|
  f.each_line do |línea|
    puts "Leí: #{línea.strip}"
  end
end

# También puedes usar File.foreach, más corto para este caso:
File.foreach("datos.txt") { |línea| puts "Leí: #{línea.strip}" }
```

El error clásico y doloroso: usar "w" cuando querías "a", borrando una bitácora o datos valiosos. Memotécnica: **"w" de "wipe" (borrar), "a" de "agregar"**. Ante la duda, usa "a". Y si vas a usar "w", asegúrate de que realmente quieres empezar de cero.

Otro detalle: los métodos ``puts``, ``print`` y ``write`` funcionan sobre el objeto archivo igual que en consola, pero escriben al archivo. ``puts`` agrega salto de línea al final; ``write`` no.

Leer archivos de varias formas

Según lo que necesites, elige la herramienta:

```
# Todo el contenido como un solo string
texto = File.read("notas.txt")

# Todas las líneas como un arreglo
líneas = File.readlines("notas.txt")
puts "El archivo tiene #{líneas.length} líneas"
puts "Primera línea: #{líneas.first.strip}"

# Línea por línea sin cargar todo (para archivos grandes)
File.foreach("notas.txt") do |línea|
  puts línea.upcase # procesar una a la vez
end

# Leer con open y each_line (equivalente a foreach)
File.open("notas.txt") do |f|
  f.each_line.with_index do |línea, i|
    puts "#{i + 1}: #{línea.strip}"
  end
end
```

Regla práctica: archivos pequeños (menos de unos MB), `File.read` o `File.readlines` sin culpa. Archivos grandes o de tamaño desconocido (logs, dumps), `File.foreach` línea por línea para no saturar la memoria.

Siempre que abras un archivo que el usuario te indicó (o que puede no existir), protege con `rescue`, porque `File.read` de un archivo inexistente lanza `Errno::ENOENT`:

```
def leer_config(ruta)
  File.read(ruta)
rescue Errno::ENOENT
  puts "No encontré #{ruta}, usando configuración por defecto"
  "tema=claro\nidioma=es\n"
end
```

Dir: trabajar con directorios enteros

Los programas reales no manejan un archivo: manejan carpetas llenas. La clase `Dir` te permite listar, recorrer y crear directorios:

```
# Listar todo en una carpeta
puts Dir.entries(".").inspect
# [".", "..", "notas.txt", "bitacora.txt", ...]
# Ojo: incluye "." (actual) y ".." (padre), casi siempre los querrás filtrar

# Listar solo archivos con cierta extensión (glob)
puts Dir.glob("*.txt").inspect      # todos los .txt de la carpeta actual
puts Dir.glob("**/*.rb").inspect    # todos los .rb, incluyendo subcarpetas

# Recorrer y procesar
Dir.glob("*.txt").each do |archivo|
  lineas = File.readlines(archivo).length
  puts "#{archivo}: #{lineas} líneas"
end

# Crear directorios
Dir.mkdir("reportes") unless Dir.exist?("reportes")

# Recorrer con bloque (como File.open, se cierra solo)
Dir.open(".") do |d|
  d.each { |entrada| puts entrada }
end
```

`Dir.glob` con patrones es una joya: `*` significa "cualquier cosa", `**` **significa "cualquier nivel de subcarpetas", y puedes combinar: `Dir.glob("datos/*.csv")`** encuentra todos los CSV dentro de datos y sus subcarpetas. Es la herramienta para "procesa todos los archivos que cumplan X".

Un programa típico de procesamiento por lotes junta Dir y File:

```
# Contar líneas de todos los archivos Ruby del proyecto
total = 0
Dir.glob("**/*.rb").each do |archivo|
  lineas = File.foreach(archivo).count
  puts "#{archivo}: #{lineas} líneas"
  total += lineas
end
puts "Total: #{total} líneas de código"
```

Error común: olvidar filtrar "." y ".." cuando usas `Dir.entries` o `Dir.open`, y luego intentar abrirlos como archivos. Con `Dir.glob` no tienes ese problema porque solo devuelve coincidencias reales.

Time.now: la hora actual y operaciones con tiempo

`Time.now` te da la fecha y hora actuales como un objeto `Time`. Y los objetos `Time` saben hacer aritmética: puedes sumarlos, restarlos y compararlos.

```
ahora = Time.now
puts ahora
# 2026-09-19 10:30:00 -0600 (fecha, hora y zona horaria)

# Los Time se pueden comparar
ayer = Time.now - 86400 # 86400 segundos = 1 día
puts ayer < ahora # true

# Sumar/restar segundos
en_una_hora = Time.now + 3600
hace_media_hora = Time.now - 1800

# Diferencia entre dos tiempos (en segundos)
inicio = Time.now
# ... código que tarda ...
fin = Time.now
puts "Tardó #{fin - inicio} segundos"
```

La aritmética de Time es en SEGUNDOS: + 3600 es una hora más, - 86400 es un día menos. Para cálculos legibles, define constantes: SEGUNDOS_POR_DIA = 86400, o mejor aún, usa la biblioteca Date para operaciones en días y meses (la vemos abajo).

Métodos útiles del objeto Time:

```
t = Time.now
puts t.year      # 2026
puts t.month     # 9
puts t.day       # 19
puts t.hour      # 10
puts t.min       # 30
puts t.sec       # 15
puts t.wday      # 6 (0=domingo, 6=sábado)
puts t.monday?   # false
puts t.sunday?   # false
```

Date: fechas calendario sin la hora

Cuando solo te importa la fecha (cumpleaños, vencimientos, calendarios), la clase Date es más cómoda que Time. Vive en la biblioteca estándar `date`, así que hay que pedirla con require:

```
require "date"

hoy = Date.today
puts hoy # 2026-09-19

# Aritmética en DÍAS (más legible que segundos)
puts hoy + 7 # una semana después
puts hoy - 30 # hace un mes (aprox)
puts hoy >> 1 # un mes después (>> suma meses)
puts hoy << 2 # dos meses antes (<< resta meses)

# Comparaciones
navidad = Date.new(2026, 12, 25)
puts navidad > hoy # true
puts (navidad - hoy).to_i # días que faltan (Date resta en días, no segundos)

# Día de la semana
puts hoy.strftime("%A") # Saturday (en inglés por defecto)
```


Fíjate en la diferencia clave: Date trabaja en DÍAS de forma natural (hoy + 7 es una semana), mientras que Time trabaja en segundos. Para "la fecha de vencimiento es en 30 días", Date es tu herramienta. Para "el proceso tardó 2.5 segundos", Time es la tuya.

También existe DateTime (fecha + hora) en la misma biblioteca, pero en la práctica Time cubre casi todos los casos de fecha-hora, y Date cubre los casos de solo-fecha.

strftime: formatear fechas como profesional

Los objetos Time y Date se muestran en un formato técnico por defecto (2026-09-19 10:30:00 -0600), pero a los humanos les gusta ver "19 de septiembre de 2026, 10:30 AM". Para eso está `strftime` (string format time): le pasas una plantilla con códigos y te devuelve el string formateado.

Los códigos que más vas a usar:

- %Y : año con 4 dígitos (2026)
- %y : año con 2 dígitos (26)
- %m : mes con 2 dígitos (09)
- %d : día con 2 dígitos (19)
- %H : hora 24h (10)
- %I : hora 12h (10)
- %M : minutos (30)
- %S : segundos (15)
- %p : AM/PM
- %A : día de la semana completo (Saturday)
- %a : día abreviado (Sat)
- %B : mes completo (September)
- %b : mes abreviado (Sep)

```
t = Time.new(2026, 9, 19, 10, 30, 0)

puts t.strftime("%Y-%m-%d")           # 2026-09-19 (formato ISO, ideal para
archivos)
puts t.strftime("%d/%m/%Y")           # 19/09/2026 (formato latino común)
puts t.strftime("%H:%M:%S")           # 10:30:00
puts t.strftime("%I:%M %p")           # 10:30 AM
puts t.strftime("%A %d de %B de %Y")   # Saturday 19 de September de 2026
```

El formato ISO (%Y-%m-%d) merece mención especial: es el estándar internacional, ordena alfabéticamente igual que cronológicamente (los archivos "2026-09-19.txt" se ordenan solos por fecha), y no es ambiguo como 09/10/2026 (¿9 de octubre o 10 de septiembre?). Para nombres de archivo, bitácoras y bases de datos, usa SIEMPRE el formato ISO.

Los nombres de días y meses salen en inglés por defecto. Si los quieres en español, la forma simple es un arreglo de traducción:

```
MESES = %w[enero febrero marzo abril mayo junio julio agosto septiembre octubre
noviembre diciembre]
DIAS = %w[domingo lunes martes miércoles jueves viernes sábado]

t = Time.now
fecha_es = "#{DIAS[t.wday]} #{t.day} de #{MESES[t.month - 1]} de #{t.year}"
puts fecha_es # sábado 19 de septiembre de 2026
```

`t.wday` da 0-6 (domingo=0) y `t.month` da 1-12, por eso el mes lleva - 1 para indexar el arreglo. Es un truco simple que resuelve la localización sin gemas externas.

Error común: confundir %m (mes) con %M (minutos) y %d (día) con... bueno, no hay otro %d, pero %m vs %M es la trampa clásica. Mayúsculas y minúsculas importan: %M son minutos, %m es mes. Si tu fecha sale con el mes donde deberían ir los minutos, ya sabes qué revisar.

Medir tiempo de ejecución: ¿qué tan rápido es mi código?

Una de las aplicaciones más prácticas de Time es medir cuánto tarda tu código. El patrón es simple: toma el tiempo antes, corre el código, toma el tiempo después, resta:

```

inicio = Time.now
# ... código a medir ...
100000.times { |i| i * i }
fin = Time.now
puts "Tardó #{fin - inicio} segundos"

```

Para mediciones más precisas existe ``Process.clock_gettime(Process::CLOCK_MONOTONIC)``, que usa un reloj monotónico (no se ve afectado si el usuario cambia la hora del sistema a la mitad de la medición). En la práctica, para tus ejercicios `Time.now` basta; el reloj monotónico es para benchmarks serios:

```

def medir(nombre)
  inicio = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  resultado = yield
  fin = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  puts "#{nombre}: #{(fin - inicio).round(4)} segundos"
  resultado
end

medir("concatenar") { 10000.times.map { |i| "n#{i}" }.join(",") }
medir("interpoliar") { 10000.times.map { |i| "n#{i}" }.join(",") }

```

Este patrón de ``medir`` con bloque (`yield`) es reutilizable: envuelves cualquier código y te dice cuánto tardó. Úsalo cuando tengas dos formas de resolver algo y quieras saber cuál es más rápida con datos reales en lugar de suposiciones.

Una advertencia honesta sobre optimización: no midas ni optimices por deporte. Primero escribe código claro y correcto; mide solo cuando algo se sienta lento o cuando tengas que elegir entre alternativas. La legibilidad casi siempre vale más que unos milisegundos.

Buenas prácticas al juntar errores, archivos y tiempo

1. **Rescata lo específico, registra lo inesperado.** Cada rescue debe saber exactamente qué error maneja. Un rescue genérico al final como red de seguridad está bien, pero debe REGISTRAR el error (clase, mensaje, backtrace), nunca tragarlo en silencio.
2. **Los archivos se abren con bloque.** `File.open` con bloque cierra solo. Si ves un `File.open` sin bloque en tu código, pregúntate quién lo cierra y cuándo.
3. **"w" borra, "a" agrega.** Ante la duda usa "a". Si usas "w", que sea una decisión consciente.

4. **Valida al inicio con raise.** Tus métodos deben fallar rápido con mensajes claros en lugar de dejar que los datos malos viajen hasta quién sabe dónde.
5. **Fechas en formato ISO para máquinas, formato amable para humanos.** Archivos, bitácoras y bases de datos: %Y-%m-%d %H:%M:%S. Pantalla para el usuario: el formato que le resulte natural.
6. **No rescates Exception.** Rescata StandardError o algo más específico. Deja que SystemExit y las señales del sistema hagan su trabajo.

CSV: el formato de intercambio universal

Hay un formato de archivo que vas a encontrar en casi todos los trabajos: el CSV (valores separados por comas). Excel lo abre, las bases de datos lo exportan, las APIs lo devuelven. Ruby trae la biblioteca `csv` en su estándar, y usarla bien te evita el error clásico de partir líneas con `split(",")` a mano (que se rompe en cuanto un campo contiene una coma entre comillas).

```
require "csv"

# Leer un CSV con encabezados: cada fila es como un hash
CSV.foreach("ventas.csv", headers: true) do |fila|
  puts "#{fila['producto']}: ${fila['monto']} (${fila['fecha']})"
end

# Escribir un CSV con encabezados
CSV.open("reporte.csv", "w") do |csv|
  csv << ["producto", "monto", "fecha"] # encabezados
  csv << ["Laptop", 15000, "2026-09-19"]
  csv << ["Mouse", 300, "2026-09-19"]
end

# Convertir un CSV completo a arreglo de hashes
datos = CSV.read("ventas.csv", headers: true).map(&:to_h)
puts datos.inspect
```

Con `headers: true`, cada fila se comporta como un hash donde las claves son los encabezados: `fila['producto']`. Eso hace el código legible y resistente a que alguien reordene las columnas (con `split(",")[0]` a mano, un reordenamiento rompe todo en silencio).

¿Por qué no usar `split(",")`? Porque el CSV real permite comas dentro de campos si van entre comillas: `"Pérez, Juan",30,"Ciudad de México"`. Un `split` ingenuo partiría "Pérez" y " Juan" como campos separados y corrompería la fila. La biblioteca csv maneja comillas, saltos de

línea dentro de campos y escapes correctamente. Regla simple: si el archivo se llama .csv, usa la biblioteca csv, no split.

Un ejemplo completo: leer ventas, sumar por producto y escribir un resumen:

```
require "csv"

# Crear datos de ejemplo
CSV.open("ventas.csv", "w") do |csv|
  csv << ["producto", "monto"]
  csv << ["Laptop", 15000]
  csv << ["Mouse", 300]
  csv << ["Laptop", 14500]
  csv << ["Teclado", 800]
  csv << ["Mouse", 350]
end

# Procesar: sumar montos por producto
totales = Hash.new(0)
CSV.foreach("ventas.csv", headers: true) do |fila|
  totales[fila["producto"]] += fila["monto"].to_i
end

# Escribir resumen ordenado de mayor a menor
CSV.open("resumen.csv", "w") do |csv|
  csv << ["producto", "total_vendido"]
  totales.sort_by { |_, total| -total }.each do |producto, total|
    csv << [producto, total]
  end
end

puts File.read("resumen.csv")
```

Este programita (leer, agregar, escribir) es el 80% de lo que se hace con datos en el mundo real: reportes de ventas, migraciones entre sistemas, limpieza de bases de datos. Con la biblioteca csv, File y un hash, ya puedes hacerlo.

Ejemplos paso a paso

Ejemplo 1: Calculadora que no se muere nunca

Una calculadora de consola que usa `begin/rescue/else/ensure` para manejar todos los errores posibles: división entre cero, texto en lugar de números y operaciones desconocidas. El programa sigue pidiendo operaciones hasta que el usuario escribe "salir".

```

def calcular(a, b, operacion)
  case operacion
  when "+"
    a + b
  when "-"
    a - b
  when "*"
    a * b
  when "/"
    a / b.to_f
  else
    raise ArgumentError, "Operación '#{operacion}' no reconocida (usa + - * /)"
  end
end

puts "Calculadora (escribe 'salir' para terminar)"
loop do
  print "Operación (ej. 10 / 2): "
  entrada = gets.chomp
  break if entrada.downcase == "salir"

  begin
    partes = entrada.split
    a = Float(partes[0])
    b = Float(partes[2])
    resultado = calcular(a, b, partes[1])
  rescue ArgumentError => e
    puts "Error en los datos: #{e.message}"
  rescue ZeroDivisionError
    puts "Error: no se puede dividir entre cero"
  rescue StandardError => e
    puts "Error inesperado (#{e.class}): #{e.message}"
  else
    puts "Resultado: #{resultado}"
  end
  ensure
    puts "---"
  end
end

puts "¡Hasta luego!"

```

Prueba escribir "10 / 0", "hola + 2", "5 ^ 3" y "8 * 7" para ver cada rama en acción. El `Float()` es estricto: lanza `ArgumentError` si el texto no es un número válido (a diferencia de `to_f`, que devuelve 0.0 en silencio). Para una calculadora, lo estricto es mejor: prefieres un error claro a un 0.0 misterioso.

Ejemplo 2: Lector de archivo con manejo de errores reales

Un programa que pide al usuario la ruta de un archivo y lo muestra numerado por líneas. Maneja: archivo inexistente, sin permisos y errores inesperados. Además crea un archivo de ejemplo si no existe, para que puedas probarlo de inmediato.

```
# Crear archivo de ejemplo para probar
File.write("ejemplo.txt", "Primera línea\nSegunda línea\nTercera línea\n") unless
File.exist?("ejemplo.txt")

def mostrar_archivo(ruta)
  lineas = File.readlines(ruta)
  puts "El archivo tiene #{lineas.length} líneas:"
  lineas.each_with_index do |linea, i|
    puts "#{i + 1}: #{linea.strip}"
  end
rescue Errno::ENOENT
  puts "No existe el archivo '#{ruta}'. Revisa la ruta."
rescue Errno::EACCES
  puts "Sin permiso para leer '#{ruta}'."
rescue StandardError => e
  puts "No se pudo leer el archivo: #{e.class}: #{e.message}"
end

print "Ruta del archivo (prueba con ejemplo.txt): "
ruta = gets.chomp
mostrar_archivo(ruta)
mostrar_archivo("no_existe.txt") # para ver el manejo del error
```

Este es el patrón que usarás cada vez que tu programa dependa de un archivo externo: intenta leer, y si falla, explica QUÉ pasó en lenguaje humano en lugar de dejar que el stack trace asuste al usuario.

Ejemplo 3: Bitácora con modo append y errores propios

Un mini sistema de bitácora que escribe eventos con fecha y hora en un archivo, usando "a" para no borrar nada. Incluye una clase de error propia para mensajes vacíos y demuestra ensure cerrando recursos incluso cuando algo falla.


```

class MensajeVacio < StandardError; end

class Bitacora
  def initialize(ruta)
    @ruta = ruta
  end

  def registrar(evento)
    raise MensajeVacio, "El evento no puede estar vacío" if evento.strip.empty?

    File.open(@ruta, "a") do |f|
      f.puts "#{Time.now.strftime('%Y-%m-%d %H:%M:%S')} | #{evento}"
    end
    puts "Registrado: #{evento}"
  rescue MensajeVacio => e
    puts "No se registró nada: #{e.message}"
  rescue SystemCallError => e
    puts "Error del sistema al escribir: #{e.message}"
  end

  def mostrar
    puts "--- Contenido de #{@ruta} ---"
    puts File.read(@ruta)
  rescue Errno::ENOENT
    puts "Aún no hay bitácora."
  end
end

b = Bitacora.new("app.log")
b.registrar("Inicio del programa")
b.registrar("Usuario Ana entró al sistema")
b.registrar(" ") # lanza MensajeVacio, se maneja con elegancia
b.registrar("Proceso terminado")
b.mostrar

```

SystemCallError es la madre de los errores del sistema operativo (permisos, disco lleno, etc.): rescatarla cubre toda esa familia de un golpe. Y fíjate que el modo "a" permite ejecutar el programa cien veces sin perder registros anteriores: cada corrida agrega sus líneas al final.

Ejemplo 4: Copiador de archivos con validaciones

Un programa que copia un archivo a otro lado, validando todo antes de tocar nada: que el origen exista, que el destino no sea el mismo, y confirmando antes de sobrescribir. Usa raise para las validaciones y rescue para los errores del sistema.

```
class MismoArchivo < StandardError; end

def copiar(origen, destino)
  raise MismoArchivo, "Origen y destino son el mismo" if File.expand_path(origen)
  == File.expand_path(destino)
  raise Errno::ENOENT, "No existe #{origen}" unless File.exist?(origen)

  if File.exist?(destino)
    print "'#{destino}' ya existe. ¿Sobrescribir? (s/n): "
    return puts "Copia cancelada." unless gets.chomp.downcase == "s"
  end

  contenido = File.read(origen)
  File.write(destino, contenido)
  puts "Copiado: #{origen} -> #{destino} (#{contenido.length} caracteres)"
  rescue MismoArchivo, Errno::ENOENT => e
    puts "No se pudo copiar: #{e.message}"
  rescue SystemCallError => e
    puts "Error del sistema: #{e.message}"
  end

  # Prueba: crea un archivo y cópialo
  File.write("original.txt", "Contenido importante\nLínea 2\n")
  copiar("original.txt", "copia.txt")
  copiar("original.txt", "original.txt") # error: mismo archivo
  copiar("fantasma.txt", "copia2.txt")   # error: no existe
end
```

Dos técnicas nuevas aquí: `rescue MismoArchivo, Errno::ENOENT => e` atrapa VARIOS tipos en un solo rescue (cuando el manejo es el mismo, no repitas código), y `File.expand\_path` convierte rutas relativas a absolutas para comparar correctamente (porque `./original.txt` y `original.txt` son el mismo archivo aunque el texto difiera).

Ejemplo 5: Procesador de carpeta completa con Dir.glob

Un programa que recorre todos los archivos .txt de una carpeta (incluyendo subcarpetas), cuenta palabras de cada uno y genera un reporte. Es el ejemplo arquetípico de "automatiza una tarea aburrida".

```

require "fileutils"

# Preparar datos de ejemplo
FileUtils.mkdir_p("documentos/sub")
File.write("documentos/nota1.txt", "Ruby es genial\nMe encanta programar\n")
File.write("documentos/nota2.txt", "Otra nota corta\n")
File.write("documentos/sub/nota3.txt", "Nota en subcarpeta con más palabras aquí\n")

total_palabras = 0
total_archivos = 0
reporte_lineas = []

Dir.glob("documentos/**/*.txt").each do |ruta|
  begin
    contenido = File.read(ruta)
    palabras = contenido.split.length
    total_palabras += palabras
    total_archivos += 1
    reporte_lineas << "#{ruta}: #{palabras} palabras"
  rescue SystemCallError => e
    reporte_lineas << "#{ruta}: ERROR (#{e.message})"
  end
end

reporte = <<~REPORTE
REPORTE DE DOCUMENTOS
=====
#{reporte_lineas.join("\n")}

Total de archivos: #{total_archivos}
Total de palabras: #{total_palabras}
Generado: #{Time.now.strftime('%Y-%m-%d %H:%M')}
REPORTE

File.write("reporte.txt", reporte)
puts reporte

```

Junta `Dir.glob` (para encontrar archivos), `File.read` (para leerlos), `begin/rescue` DENTRO del `each` (para que un archivo dañado no detenga el procesamiento de los demás), `heredoc` (para el reporte) y `File.write` (para guardarlo). Este esqueleto te sirve para cualquier tarea por lotes: renombrar archivos, convertir formatos, buscar texto en muchos archivos.

Ejemplo 6: Agenda con fechas usando Date

Una mini agenda que guarda eventos con fecha, avisa cuáles son hoy, cuáles ya pasaron y cuántos días faltan para los próximos. Usa Date para la aritmética en días.

```
require "date"

class Agenda
  def initialize
    @eventos = []
  end

  def agregar(descripcion, fecha_str)
    fecha = Date.parse(fecha_str) # acepta "2026-09-25", "25/09/2026", etc.
    @eventos << { descripcion: descripcion, fecha: fecha }
    puts "Agregado: #{descripcion} (#{fecha.strftime('%d/%m/%Y')})"
    rescue Date::Error => e
      puts "Fecha inválida '#{fecha_str}': #{e.message}"
    end

  def mostrar
    hoy = Date.today
    puts "\n=== AGENDA (hoy: #{hoy.strftime('%d/%m/%Y')}) ==="
    @eventos.sort_by { |e| e[:fecha] }.each do |e|
      dias = (e[:fecha] - hoy).to_i
      estado =
        if dias < 0
          "hace #{-dias} días"
        elsif dias == 0
          "¡HOY!"
        else
          "en #{dias} días"
        end
      puts "- #{e[:descripcion]}: #{e[:fecha].strftime('%d/%m/%Y')} (#{estado})"
    end
  end
end

agenda = Agenda.new
agenda.agregar("Examen de Ruby", (Date.today + 7).to_s)
agenda.agregar("Cumpleaños de Ana", (Date.today - 3).to_s)
agenda.agregar("Entrega de proyecto", Date.today.to_s)
agenda.agregar("Fecha mala", "no-es-fecha")
agenda.mostrar
```

`Date.parse` es flexible con formatos pero lanza `Date::Error` si no entiende el texto: por eso el `rescue`. La resta de fechas da un número racional de días (por eso el `.to_i`). Y ``sort_by`` ordena los eventos cronológicamente. Con este ejemplo ya puedes construir recordatorios, vencimientos y calendarios.

Ejemplo 7: Medidor de rendimiento comparando dos algoritmos

Compara dos formas de construir un string largo: concatenar con `+` en un loop vs usar `map+join`. Mide con el reloj monotónico para precisión.

```
def medir(nombre, repeticiones = 3)
  tiempos = repeticiones.times.map do
    inicio = Process.clock_gettime(Process::CLOCK_MONOTONIC)
    yield
    Process.clock_gettime(Process::CLOCK_MONOTONIC) - inicio
  end
  promedio = tiempos.sum / tiempos.length
  puts "#{nombre}: #{promedio.round(4)} segundos (promedio de #{repeticiones})"
end

n = 20000

medir("Concatenar con +") do
  resultado = ""
  n.times { |i| resultado += "palabra#{i} " }
end

medir("Map + join") do
  resultado = n.times.map { |i| "palabra#{i} " }.join
end

medir("Push + join") do
  partes = []
  n.times { |i| partes << "palabra#{i} " }
  resultado = partes.join
end
```

Al correrlo verás que concatenar con `+` es notablemente más lento: cada `+=` crea un string nuevo y copia todo lo anterior. Las versiones con `join` construyen el resultado de una sola vez. Por eso la recomendación: para armar strings grandes en un loop, junta las partes en un arreglo y haz `join` al final. Y nota que medir varias veces y promediar da resultados más confiables que una sola medición.

Ejemplo 8: Registro de usuarios con archivo, fechas y errores

El ejemplo integrador: un sistema que registra usuarios en un archivo CSV (nombre,email,fecha_registro), valida el email con regex, evita duplicados y muestra el registro con antigüedad en días. Junta archivos, excepciones, regex, Time y Date.

```
require "date"

ARCHIVO = "usuarios.csv"
EMAIL_OK = /\A[^\s]+@[^\s]+\.[^\s]+\z/

class EmailInvalido < StandardError; end
class UsuarioDuplicado < StandardError; end

def registrar_usuario(nombre, email)
  email = email.strip.downcase
  raise EmailInvalido, "'#{email}' no parece un email válido" unless
    email.match?(EMAIL_OK)

  if File.exist?(ARCHIVO)
    duplicado = File.foreach(ARCHIVO).any? { |linea| linea.split(",")[1]&.strip ==
      email }
    raise UsuarioDuplicado, "El email #{email} ya está registrado" if duplicado
  end

  File.open(ARCHIVO, "a") do |f|
    f.puts "#{nombre.strip},#{email},#{Date.today}"
  end
  puts "Registrado: #{nombre} (#{email})"
rescue EmailInvalido, UsuarioDuplicado => e
  puts "No se registró: #{e.message}"
rescue SystemCallError => e
  puts "Error de archivo: #{e.message}"
end

def mostrar_usuarios
  puts "\n=== USUARIOS REGISTRADOS ==="
  File.foreach(ARCHIVO).with_index do |linea, i|
    nombre, email, fecha_str = linea.strip.split(",")
    dias = (Date.today - Date.parse(fecha_str)).to_i
    puts "#{i + 1}. #{nombre} (#{email}) - registrado hace #{dias} días"
  end
rescue Errno::ENOENT
  puts "Aún no hay usuarios registrados."
end
```

```
# Pruebas
File.delete(ARCHIVO) if File.exist?(ARCHIVO)
registrar_usuario("Ana López", "ana@correo.com")
registrar_usuario("Beto Ruiz", "beto@correo.com")
registrar_usuario("Otro Ana", "ana@correo.com") # duplicado
registrar_usuario("Malo", "no-es-email")        # inválido
mostrar_usuarios
```

Cada pieza de la unidad aparece: clases de error propias, validación con raise, regex para el email, modo "a" para no borrar registros, File.foreach para leer sin cargar todo, Date para la antigüedad, y rescates específicos para cada situación. Si entiendes este programa completo, entiendes la unidad.

Ejemplo 9: ensure para limpieza garantizada

Demuestra que ensure se ejecuta SIEMPRE: con éxito, con excepción rescatada, con excepción no rescatada e incluso con return en medio. Es la garantía de limpieza de Ruby.

```

def experimento(caso)
  puts "--- Caso: #{caso} ---"
  begin
    puts "Abriendo recurso..."
    case caso
    when :ok
      puts "Todo bien"
    when :rescatado
      raise "fallo manejable"
    when :no_rescatado
      raise "fallo grave"
    when :return_temprano
      return "me voy temprano"
    end
  rescue RuntimeError => e
    puts "Rescatado: #{e.message}" if caso == :rescatado
    raise if caso == :no_rescatado # relanzo: no lo puedo manejar
  ensure
    puts "Cerrando recurso (SIEMPRE)"
  end
  puts "Fin normal del método"
end

experimento(:ok)
experimento(:rescatado)
experimento(:return_temprano)
begin
  experimento(:no_rescatado)
rescue RuntimeError => e
  puts "Atrapado afuera: #{e.message}"
end

```

Observa la salida con atención: en los cuatro casos aparece "Cerrando recurso (SIEMPRE)". Incluso con return temprano y con excepción relanzada, el ensure corrió. Por eso la limpieza de recursos (cerrar archivos, conexiones, etc.) va en ensure: es el único lugar con garantía total de ejecución.

Ejemplo 10: Reloj de cuenta regresiva con Time

Un temporizador que muestra una cuenta regresiva en consola usando Time para calcular cuánto falta. Simple, visual y útil.


```

def cuenta_regresiva(segundos)
  raise ArgumentError, "Los segundos deben ser positivos" if segundos <= 0

  fin = Time.now + segundos
  loop do
    restante = (fin - Time.now).ceil
    break if restante <= 0
    print "\rQuedan #{restante} segundos... "
    sleep 1
  end
  puts "\r¡Tiempo! #{" " * 20}"
rescue Interrupt
  puts "\nCuenta cancelada por el usuario."
end

puts "Cuenta regresiva de 5 segundos:"
cuenta_regresiva(5)

```

Detalles a notar: ``r`` (retorno de carro) regresa el cursor al inicio de la línea para actualizar el número sin hacer scroll; ``ceil`` redondea hacia arriba para no mostrar "0 segundos" antes de tiempo; y ``rescue Interrupt`` atrapa el Ctrl+C (Interrupt cuelga de `SignalException...` en realidad `Interrupt` hereda de `SignalException` que hereda de `Exception`, así que un `rescue` a secas NO lo atraparía; hay que pedirlo explícito). Ese último punto es sutil e importante: las señales del sistema no se atrapan por accidente, solo cuando las pides.

Bitácoras con rotación: que el log no crezca hasta el infinito

Un detalle que separa un programa de juguete de uno profesional: las bitácoras crecen sin parar. Un programa que escribe 100 líneas por minuto genera más de 140,000 líneas al día. Si nunca rotas el archivo, un día el disco se llena y TODO falla. La rotación es simple: cuando el archivo supera cierto tamaño, lo renombras (`app.log -> app.log.1`) y empiezas uno nuevo.

Ruby trae esto resuelto en la biblioteca estándar ``logger``, que además agrega niveles de severidad (`debug`, `info`, `warn`, `error`, `fatal`) y rotación automática:

```
require "logger"

# Logger con rotación: guarda hasta 3 archivos de 1 MB (app.log, app.log.0,
app.log.1...)
log = Logger.new("app.log", 3, 1024 * 1024)
log.level = Logger::INFO # ignora los debug en producción

log.debug("Detalle técnico que solo importa depurando")
log.info("El programa inició correctamente")
log.warn("El disco está al 85% de capacidad")
log.error("No se pudo conectar a la base de datos")
log.fatal("Error crítico: cerrando el programa")

# Formato personalizado con fecha ISO
log.formatter = proc do |severidad, tiempo, _prog, mensaje|
  "#{tiempo.strftime('%Y-%m-%d %H:%M:%S')} [{severidad}] #{mensaje}\n"
end
log.info("Con formato personalizado")
```

Los niveles de severidad son un filtro: en desarrollo pones el nivel en DEBUG y ves todo; en producción lo pones en INFO o WARN y solo ves lo importante. Cada mensaje lleva su nivel, así que puedes buscar "ERROR" en el archivo y encontrar los problemas sin leer miles de líneas.

Los parámetros de `Logger.new("app.log", 3, 1024\*1024)` significan: guarda hasta 3 archivos viejos, cada uno de máximo 1 MB. Cuando app.log supera 1 MB, se renombra a app.log.0 (y el viejo .0 pasa a .1, etc.) y se crea un app.log nuevo. Automático, sin que escribas una línea de rotación.

¿Cuándo usar Logger y cuándo un simple File.open con "a"? Para scripts y ejercicios, el File.open con modo "a" que ya dominas es suficiente. Para programas que corren mucho tiempo (servidores, demonios, tareas programadas), usa Logger: la rotación y los niveles pagan su aprendizaje en la primera madrugada que algo falle y necesites encontrar el error rápido entre millones de líneas.

Práctica

Ejercicio 1: Conversor seguro de temperaturas

Escribe un método `a_celsius` que reciba un string con grados Fahrenheit y lo convierta a Celsius con la fórmula $(f - 32) * 5.0/9$. Debe manejar: texto que no es número (`ArgumentError` con `Float`), y temperaturas bajo el cero absoluto (-459.67 F) lanzando tu propia excepción `TemperaturaImposible`. Pruébalo con "212", "abc" y "-500".

Solución explicada:

```
class TemperaturaImposible < StandardError; end

def a_celsius(f_str)
  f = Float(f_str)
  raise TemperaturaImposible, "#{f}°F está bajo el cero absoluto (-459.67°F)" if f
  < -459.67
  ((f - 32) * 5.0 / 9).round(2)
rescue ArgumentError
  puts "'#{f_str}' no es un número válido"
  nil
rescue TemperaturaImposible => e
  puts "Error físico: #{e.message}"
  nil
end

puts a_celsius("212").inspect    # 100.0
puts a_celsius("abc").inspect    # mensaje + nil
puts a_celsius("-500").inspect   # mensaje + nil
puts a_celsius("32").inspect     # 0.0
```

`Float()` es estricto y lanza `ArgumentError` con texto inválido, perfecto para validar. La excepción propia distingue un error de física de un error de formato: quien use tu método puede rescatar cada uno por separado. Devolver nil en los rescates es una decisión de diseño documentada: "si no se pudo convertir, no hay resultado".

Ejercicio 2: División con reintentos

Escribe un programa que pida dos números al usuario y los divida. Si el usuario escribe algo inválido o el divisor es cero, debe mostrar el error y VOLVER A PEDIR los datos (máximo 3 intentos). Usa `begin/rescue` dentro de un loop con contador.

Solución explicada:

```

intentos = 0
maximos = 3
resultado = nil

while intentos < maximos && resultado.nil?
  intentos += 1
  begin
    print "Dividendo: "
    a = Float(gets.chomp)
    print "Divisor: "
    b = Float(gets.chomp)
    resultado = a / b
  rescue ArgumentError
    puts "Eso no era un número. Intento #{intentos} de #{maximos}."
  rescue ZeroDivisionError
    puts "No se puede dividir entre cero. Intento #{intentos} de #{maximos}."
  else
    puts "Resultado: #{resultado}"
  end
end

puts "Se acabaron los intentos." if resultado.nil?

```

El loop con la condición `resultado.nil?` es el truco: mientras no haya resultado válido, sigue intentando; el `else` solo corre cuando no hubo excepción, así que el éxito se registra ahí. El límite de intentos evita un loop infinito si el usuario insiste en escribir mal. Este patrón de "reintentar con límite" aparece en conexiones de red, lectura de sensores y cualquier entrada humana.

Ejercicio 3: Diario personal en archivo

Crea un programa que: 1) agregue una entrada al archivo "diario.txt" con fecha y hora (modo "a"), 2) muestre todas las entradas numeradas, 3) cuente cuántas entradas hay. Todo con manejo de errores si el archivo no existe al mostrar.

Solución explicada:

```

DIARIO = "diario.txt"

def agregar_entrada(texto)
  if texto.strip.empty?
    puts "No se guardó: entrada vacía."
    return
  end
  File.open(DIARIO, "a") do |f|
    f.puts "#{Time.now.strftime('%Y-%m-%d %H:%M')} - #{texto.strip}"
  end
  puts "Entrada guardada."
rescue SystemCallError => e
  puts "No se pudo guardar: #{e.message}"
end

def mostrar_diario
  lineas = File.readlines(DIARIO)
  puts "Tienes #{lineas.length} entradas:"
  lineas.each_with_index { |l, i| puts "#{i + 1}. #{l.strip}" }
rescue Errno::ENOENT
  puts "El diario está vacío (aún no existe el archivo)."
end

agregar_entrada("Hoy aprendí a manejar archivos en Ruby")
agregar_entrada("Los bloques cierran el archivo solos, qué chido")
mostrar_diario

```

El modo "a" hace que cada ejecución agregue sin borrar lo anterior: el diario crece con el tiempo. `File.readlines` en mostrar_diario lanza Errno::ENOENT si el archivo no existe (diario vacío), y el rescue lo convierte en un mensaje amable. Nota que agregar_entrada valida la entrada vacía ANTES de tocar el archivo: valida primero, actúa después.

Ejercicio 4: Buscador de texto en múltiples archivos

Escribe un método `buscar\_en\_archivos(patron, extension)` que busque un texto dentro de todos los archivos con la extensión dada en la carpeta actual (usa Dir.glob), e imprima archivo, número de línea y la línea donde aparece. Debe seguir buscando aunque un archivo falle.

Solución explicada:

```

def buscar_en_archivos(patron, extension = "txt")
  encontrados = 0
  Dir.glob("*.#{extension}").each do |ruta|
    begin
      File.foreach(ruta).with_index do |linea, i|
        if linea.include?(patron)
          puts "#{ruta}:#{i + 1}: #{linea.strip}"
          encontrados += 1
        end
      end
    rescue SystemCallError => e
      puts "#{ruta}: no se pudo leer (#{e.message}), sigo con el siguiente"
    end
  end
  puts "Total de coincidencias: #{encontrados}"
end

# Datos de prueba
File.write("a.txt", "Ruby es genial\nOtra línea\n")
File.write("b.txt", "Nada aquí\nRuby otra vez\n")

buscar_en_archivos("Ruby")

```

El begin/rescue va DENTRO del each a propósito: si un archivo falla (permisos, se borró a la mitad), el programa registra el problema y sigue con los demás en lugar de morir. `File.foreach` con `with\_index` da número de línea gratis, y el formato "archivo:línea: texto" es el mismo que usan las herramientas profesionales de búsqueda (grep, IDEs).

Ejercicio 5: Calculadora de edad exacta

Pide al usuario su fecha de nacimiento (acepta varios formatos con Date.parse) y calcula: años, meses y días vividos (aproximado con días totales), en qué día de la semana nació, y cuántos días faltan para su próximo cumpleaños. Maneja fechas inválidas y fechas futuras.

Solución explicada:

```

require "date"

class FechaFutura < StandardError; end

DIAS = %w[domingo lunes martes miércoles jueves viernes sábado]

def info_cumpleaños(fecha_str)
  nacimiento = Date.parse(fecha_str)
  hoy = Date.today
  raise FechaFutura, "Naciste en el futuro, ¿vienes del mañana?" if nacimiento >
hoy

  dias_vividos = (hoy - nacimiento).to_i
  años = (dias_vividos / 365.25).floor

  proximo = Date.new(hoy.year, nacimiento.month, nacimiento.day)
  proximo = proximo.next_year if proximo < hoy
  faltan = (proximo - hoy).to_i

  puts "Naciste un #{DIAS[nacimiento.wday]}."
  puts "Tienes aproximadamente #{años} años (#{dias_vividos} días vividos)."
  if faltan == 0
    puts "¡FELIZ CUMPLEAÑOS! ¡Hoy es el día!"
  else
    puts "Tu próximo cumpleaños es en #{faltan} días
(#{proximo.strftime('%d/%m/%Y')})."
  end
rescue Date::Error
  puts "'#{fecha_str}' no es una fecha válida. Prueba con AAAA-MM-DD."
rescue FechaFutura => e
  puts e.message
end

info_cumpleaños("2000-05-15")
info_cumpleaños("ayer")
info_cumpleaños("2050-01-01")

```

`Date.parse("ayer")`... bueno, en realidad Date.parse no entiende "ayer" (ese fue un ejemplo con trampa para que veas el rescate). Lo importante: Date.parse acepta "2000-05-15", "15/05/2000", "May 15 2000" y más. El cálculo del próximo cumpleaños maneja el caso borde de que ya pasó este año (next_year). Y la excepción propia FechaFutura convierte un dato absurdo en un mensaje con humor en lugar de un número negativo confuso.

Ejercicio 6: Renombrador masivo de archivos

Crea un programa que renombre todos los archivos .txt de una carpeta agregándoles la fecha de hoy al inicio del nombre (formato ISO). Debe: crear la carpeta "para_renombrar" con 3 archivos de ejemplo, mostrar antes/después, y manejar colisiones (si el destino ya existe, no sobrescribir).

Solución explicada:

```
require "fileutils"

FileUtils.mkdir_p("para_renombrar")
%w[notas ideas borrador].each do |n|
  File.write("para_renombrar/#{n}.txt", "contenido de #{n}\n")
end

hoy = Date.today.strftime("%Y-%m-%d")
renombrados = 0

Dir.glob("para_renombrar/*.txt").each do |ruta|
  dir = File.dirname(ruta)
  nuevo_nombre = "#{dir}/#{hoy}-#{File.basename(ruta)}"

  if File.exist?(nuevo_nombre)
    puts "OMITIDO #{ruta}: ya existe #{nuevo_nombre}"
    next
  end

  begin
    File.rename(ruta, nuevo_nombre)
    puts "#{ruta} -> #{nuevo_nombre}"
    renombrados += 1
  rescue SystemCallError => e
    puts "ERROR con #{ruta}: #{e.message}"
  end
end

puts "Renombrados: #{renombrados}"
```

`File.basename` extrae el nombre sin la carpeta y `File.dirname` lo contrario: juntos te permiten reconstruir rutas. El formato ISO en el nombre hace que los archivos se ordenen cronológicamente solos. La verificación de colisión ANTES de renombrar evita pérdidas de datos. Este tipo de script te ahorra horas cuando tienes que organizar cientos de archivos.

Ejercicio 7: Cronómetro de tareas con reporte

Escribe una clase Cronometro que permita: iniciar una tarea (guarda Time.now), detenerla (calcula duración), y al final generar un reporte en archivo con cada tarea y su duración formateada (minutos:segundos), más el total. Usa strftime o formateo manual para la duración.

Solución explicada:

```
class Cronometro
  def initialize
    @tareas = []
    @inicio_actual = nil
    @nombre_actual = nil
  end

  def iniciar(nombre)
    raise "Ya hay una tarea en curso: #{@nombre_actual}" if @inicio_actual
    @nombre_actual = nombre
    @inicio_actual = Time.now
    puts "Iniciada: #{nombre}"
  end

  def detener
    raise "No hay tarea en curso" unless @inicio_actual
    duracion = Time.now - @inicio_actual
    @tareas << { nombre: @nombre_actual, segundos: duracion }
    puts "Detenida: #{@nombre_actual} (#{formato(duracion)})"
    @inicio_actual = nil
    @nombre_actual = nil
  end

  def formato(segundos)
    minutos = (segundos / 60).floor
    segs = (segundos % 60).round
    "#{minutos}:#{segs.to_s.rjust(2, '0')}"
  end

  def reporte(ruta = "tiempos.txt")
    total = @tareas.sum { |t| t[:segundos] }
    contenido = <<~REPORTE
      REPORTE DE TIEMPOS
      #{Time.now.strftime('%Y-%m-%d %H:%M')}
      =====
      #{@tareas.map { |t| "- #{t[:nombre]}: #{formato(t[:segundos])}" }.join("\n")}

      Tiempo total: #{formato(total)}
    end
  end
end
```

```

    REPORTE
    File.write(ruta, contenido)
    puts contenido
  end
end

c = Cronometro.new
c.iniciar("Estudiar Ruby")
sleep 2
c.detener
c.iniciar("Hacer ejercicio")
sleep 1
c.detener
c.reporte

```

Los raise protegen el estado: no puedes iniciar dos tareas a la vez ni detener sin haber iniciado. El método formato convierte segundos a "m:s" legible (rjust rellena con cero a la izquierda: "2:05" en lugar de "2:5"). El reporte usa heredoc e interpolación, cerrando el círculo con la Unidad 7. Es un programa completo y útil de verdad.

Ejercicio 8: Validador de archivo de configuración

Escribe un programa que lea un archivo "config.txt" con formato clave=valor (una por línea), lo convierta en un hash, valide que existan las claves obligatorias (usuario, idioma, tema) y que el tema sea "claro" u "oscuro". Define excepciones propias: ConfigFaltante y ValorInvalido. Si el archivo no existe, usa valores por defecto y avisa.

Solución explicada:

```

class ConfigFaltante < StandardError; end
class ValorInvalido < StandardError; end

CLAVES_OBLIGATORIAS = %w[usuario idioma tema]
TEMAS_VALIDOS = %w[claro oscuro]
POR_DEFEECTO = { "usuario" => "invitado", "idioma" => "es", "tema" => "claro" }

def cargar_config(ruta = "config.txt")
  config = {}
  File.foreach(ruta) do |linea|
    linea = linea.strip
    next if linea.empty? || linea.start_with?("#")
    clave, valor = linea.split("=", 2)
    config[clave.strip] = valor.strip if clave && valor
  end
end

```

```

    faltantes = CLAVES_OBLIGATORIAS - config.keys
    raise ConfigFaltante, "Faltan claves: #{faltantes.join(', ')}" unless
faltantes.empty?

    unless TEMAS_VALIDOS.include?(config["tema"])
        raise ValorInvalido, "Tema '#{config["tema']}' inválido (usa:
#{TEMAS_VALIDOS.join(' o ')}"
    end

    puts "Configuración cargada: #{config.inspect}"
    config
rescue Errno::ENOENT
    puts "No existe #{ruta}, usando valores por defecto."
    POR_DEFECTO
rescue ConfigFaltante, ValorInvalido => e
    puts "Config inválida: #{e.message}"
    puts "Usando valores por defecto."
    POR_DEFECTO
end

# Pruebas
File.write("config.txt", "# Mi configuración\nusuario = ana\nidioma = es\ntema =
oscuro\n")
cargar_config
File.write("config.txt", "usuario = ana\nidioma = es\n")
cargar_config # falta tema
File.write("config.txt", "usuario = ana\nidioma = es\ntema = rosa\n")
cargar_config # tema inválido
File.delete("config.txt")
cargar_config # no existe: por defecto

```

Este ejercicio junta todo: lectura de archivos línea por línea, parsing manual (split con límite 2 para valores que contengan "="), comentarios con # ignorados, validación con excepciones propias, y estrategia de "valores por defecto" cuando algo falla. Es el patrón real que usan los programas para leer su configuración sin morir si el archivo está mal.

Preguntar antes de actuar: File.exist?, File.directory? y amigos

Ruby te da una familia de métodos para inspeccionar el sistema de archivos ANTES de operar, y usarlos es la diferencia entre un programa precavido y uno que vive apagando incendios con rescue. Los más útiles:

```
puts File.exist?("notas.txt")      # ¿existe? (archivo o carpeta)
puts File.file?("notas.txt")      # ¿existe Y es un archivo?
puts File.directory?("documentos") # ¿existe Y es un directorio?
puts File.size("notas.txt")        # tamaño en bytes
puts File.extname("foto.jpg")      # ".jpg"
puts File.basename("/a/b/foto.jpg") # "foto.jpg"
puts File.dirname("/a/b/foto.jpg")  # "/a/b"
puts File.zero?("vacio.txt")       # ¿existe y está vacío?
```

El patrón típico combina estos métodos con decisiones:

```
ruta = "reporte.txt"

if File.exist?(ruta) && !File.zero?(ruta)
  puts "El reporte ya existe y tiene contenido, no lo regenero."
else
  File.write(ruta, generar_reporte())
  puts "Reporte creado."
end

# Crear carpeta solo si no existe (evita el error "ya existe")
Dir.mkdir("salidas") unless Dir.exist?("salidas")
```

Una advertencia honesta de programador experimentado: entre que preguntas "¿existe?" y que actúas, el mundo puede cambiar (otro programa borra el archivo en ese instante). Es una condición de carrera clásica. Para código crítico, la estrategia robusta es intentar la operación y rescatar el error, en lugar de (o además de) preguntar antes. La regla práctica: usa `File.exist?` para decisiones de UX (mostrar mensajes, evitar sobrescribir sin preguntar), y usa `begin/rescue` para la operación real. No son excluyentes: se complementan.

Codificación de archivos: el tema del UTF-8 y los acentos

Si trabajas en español, tarde o temprano te topas con esto: lees un archivo con "niño" y te sale "niÃ±o", o escribes "José" y al abrir el archivo ves símbolos raros. Eso es un problema de CODIFICACIÓN: el archivo guarda bytes, y la codificación dice cómo interpretar esos bytes como letras.

Ruby moderno (2.0+) usa UTF-8 por defecto, que maneja acentos, ñ y emojis sin problema. La mayoría de las veces no tienes que hacer nada. Pero cuando lees archivos creados en Windows con programas viejos (codificación Windows-1252 o Latin-1), los acentos se rompen. La solución es decirle a Ruby la codificación al abrir:

```
# Leer un archivo viejo de Windows especificando su codificación
contenido = File.read("viejo.txt", encoding: "Windows-1252")
puts contenido # ahora los acentos salen bien

# Convertir a UTF-8 al leer
contenido_utf8 = File.read("viejo.txt", encoding: "Windows-1252:UTF-8")

# Forzar la escritura en UTF-8 con BOM para que Excel lo abra bien
File.write("para_excel.csv", "nombre,ciudad\nJosé,México\n", encoding: "UTF-8")
```

¿Cómo sabes qué codificación tiene un archivo? Si los acentos se ven mal al leerlo normal, prueba con "Windows-1252" (el culpable más común en archivos de Windows en español) o "ISO-8859-1". Si al abrir un archivo te sale el error "invalid byte sequence in UTF-8", es señal de que el archivo NO es UTF-8 y necesitas especificar la codificación correcta.

Buena práctica: cuando GENERES archivos que otros van a abrir (CSVs para Excel, reportes de texto), escríbelos en UTF-8 y, si van para Excel en Windows, considera agregar el BOM (marca de orden de bytes) al inicio para que Excel detecte el UTF-8 automáticamente:

```
BOM = "\uFEFF"
File.write("reporte.csv", BOM + "nombre,ventas\nAna,100\n", encoding: "UTF-8")
```

Ese BOM invisible al inicio le dice a Excel "esto es UTF-8", y tus acentos aparecen perfectos. Es un detalle pequeño que te ahorra el clásico "el CSV se ve mal en Excel".

retry: reintentar lo que falló

Hay un comando dentro de rescue que pocos principiantes conocen y que es oro en el mundo real: `retry`. Cuando lo llamas dentro de un bloque rescue, Ruby VUELVE A EJECUTAR el begin desde el inicio. Es perfecto para errores transitorios: la red falló un momento, el archivo estaba bloqueado por otro programa, el servicio no respondió a la primera.

```

intentos = 0

begin
  intentos += 1
  puts "Intento #{intentos}..."
  raise "fallo de red simulado" if intentos < 3
  puts "¡Conexión exitosa!"
rescue RuntimeError => e
  if intentos < 5
    puts "Falló (#{e.message}), reintentando en 1 segundo..."
    sleep 1
    retry
  else
    puts "Me rindo después de #{intentos} intentos"
  end
end
end

```

La regla de oro con retry es SIEMPRE poner un límite de intentos. Un retry sin límite es un loop infinito disfrazado: si el error es permanente (no transitorio), tu programa se queda reintentando para siempre. El patrón correcto es: contador, límite máximo, y una pausa (sleep) entre intentos para no saturar el servicio. En código profesional verás este patrón con "backoff exponencial" (esperar 1, 2, 4, 8 segundos...), pero la idea es la misma.

Otro detalle: las variables definidas ANTES del begin conservan su valor entre reintentos (por eso intentos funciona como contador), pero las definidas DENTRO del begin se reinician. Tenlo en cuenta si tu lógica depende de estado parcial.

Rutas absolutas vs relativas: dónde estoy parado

Cuando escribes `File.read("notas.txt")`, esa ruta es RELATIVA: Ruby la busca en el directorio de trabajo actual (donde corriste el programa, no donde está tu archivo .rb). Esto causa el clásico "en mi máquina sí funciona": corres el script desde otra carpeta y el archivo "desaparece".

```

puts Dir.pwd # directorio de trabajo actual: desde aquí se resuelven las rutas
              relativas

```

Hay dos soluciones profesionales:

1. Rutas absolutas con `File.expand_path`: convierte una ruta relativa en absoluta, resolviendo `.` y `..`:

```
puts File.expand_path("notas.txt")      # /home/ana/proyecto/notas.txt
puts File.expand_path("../otros/datos") # sube un nivel y entra a otros/datos
```

2. Rutas relativas AL ARCHIVO del programa con *dir*: *dir* es la carpeta donde vive tu archivo .rb, sin importar desde dónde lo ejecuten. Es la forma robusta de encontrar archivos "hermanos" de tu script:

```
# Estructura:
#   proyecto/
#     app.rb
#     datos/config.txt

# Dentro de app.rb, esto funciona SIEMPRE, lo ejecutes desde donde lo ejecutes:
ruta = File.join(__dir__, "datos", "config.txt")
contenido = File.read(ruta)
```

`File.join` une partes de ruta con el separador correcto del sistema operativo (/ en Linux/Mac, \ en Windows). Nunca concatenes rutas con + y "/" a mano si tu programa puede correr en Windows: File.join es portable.

Buena práctica: al inicio de tus programas, define una constante con la raíz del proyecto y construye todas las rutas desde ahí:

```
RAIZ = __dir__
CONFIG = File.join(RAIZ, "config", "app.txt")
LOGS = File.join(RAIZ, "logs")
```

Así tu programa funciona igual lo corras desde donde lo corras, y si mueves la carpeta del proyecto, todo sigue funcionando.

El backtrace: leer la pila de llamadas como detective

Cuando una excepción no es rescatada, Ruby imprime el backtrace: la lista de llamadas que llevaron al error, de la más reciente a la más antigua. Aprender a leerlo es la habilidad de depuración más rentable que puedes desarrollar.

```

def nivel3
  raise "algo salió mal"
end

def nivel2
  nivel3
end

def nivel1
  nivel2
end

begin
  nivel1
rescue RuntimeError => e
  puts "Error: #{e.message}"
  puts "Backtrace:"
  puts e.backtrace.first(5)
end

```

La salida se ve así (las rutas varían):

```

Error: algo salió mal
Backtrace:
programa.rb:2:in 'nivel3'
programa.rb:6:in 'nivel2'
programa.rb:10:in 'nivel1'
programa.rb:15:in '<main>'

```

Se lee de ARRIBA hacia abajo: la primera línea es DONDE se lanzó el error (nivel3, línea 2), y cada línea siguiente es quién llamó a quién. El 90% de las veces, la primera o segunda línea te dice exactamente dónde está el problema. Las líneas de ``<main>`` o de gemas externas suelen ser ruido: enfócate en TUS archivos.

En un programa real, cuando un usuario te reporte un error, pídele el mensaje COMPLETO con el backtrace, no solo "me salió un error". Con el backtrace reproduces el camino exacto. Y cuando registres errores en tu bitácora, guarda `e.backtrace.first(10)` junto con el mensaje: tu yo del futuro investigando un bug en producción te lo agradecerá.

Un truco final: `caller` te da el backtrace del punto actual SIN necesidad de una excepción. Sirve para depurar "¿quién me está llamando?" cuando un método se ejecuta en momentos inesperados:


```
def metodo_sospechoso
  puts "Me llamaron desde:"
  puts caller.first(3)
end
```

Resumen de la unidad

- Una excepción es un objeto-error que interrumpe el programa; si nadie la atrapa, el programa termina mostrando el stack trace.
- begin/rescue/else/ensure: protege código riesgoso, define plan B, separa el código de éxito (else) y garantiza limpieza (ensure, que corre siempre).
- Rescata errores ESPECÍFICOS (ZeroDivisionError, Errno::ENOENT, ArgumentError); un `rescue` a secas atrapa StandardError; NUNCA rescates Exception a ciegas.
- Pon los rescue del más específico al más general; puedes atrapar varios tipos en un solo rescue separándolos con comas.
- `raise` lanza tus propios errores: valida al inicio de tus métodos y escribe mensajes que digan qué está mal y qué se esperaba.
- Crea clases de error propias heredando de StandardError cuando necesites distinguir errores al atraparlos; una clase base común permite atrapar "cualquier error de mi app".
- File.read / File.write para lo simple; File.open con bloque para lo controlado (cierra solo, incluso con excepciones).
- Modos: "r" leer (por defecto), "w" escribir BORRANDO, "a" agregar al final; ante la duda usa "a".
- Para archivos grandes usa File.foreach (línea por línea) en lugar de cargar todo en memoria.
- Dir.glob con patrones (*,) **encuentra archivos por carpeta y subcarpetas; Dir.mkdir crea directorios.**
- Time.now da fecha+hora actual; su aritmética es en segundos; úsalo para medir duraciones y comparar momentos.
- Date (require "date") es para solo-fechas con aritmética en días; Date.parse entiende varios formatos.
- strftime formatea fechas: %Y-%m-%d es el ISO (úsalo en archivos y bitácoras), %d/%m/%Y para humanos; cuidado con %m (mes) vs %M (minutos).

- Para medir rendimiento: `Time.now` antes/después, o `Process.clock_gettime(Process::CLOCK_MONOTONIC)` para precisión; mide varias veces y promedia.
- Reglas de oro: valida primero con `raise`, rescata específico, registra lo inesperado, abre archivos con bloque, y nunca silencias un error sin registrarlo.

UNIDAD 9: Gemas, pruebas y regex

Ya llegaste lejos. A estas alturas sabes crear clases, heredar, manejar archivos, usar módulos y hasta conectar tu programa con la realidad mediante JSON. Pero si te quedas solo con lo que Ruby trae de fábrica, estás usando apenas una parte del poder del lenguaje. Esta unidad te abre las tres puertas que separan al programador principiante del programador productivo: las gemas, las pruebas y las expresiones regulares.

Las gemas son paquetes de código que otros programadores ya escribieron y empaquetaron para que tú los reutilices. ¿Necesitas colores en la terminal? Hay una gema. ¿Necesitas datos falsos para probar tu app? Hay una gema. ¿Necesitas consultar una API por HTTP? Hay una gema. La comunidad de Ruby es famosa precisamente por esto: cuando alguien resuelve un problema, lo empaqueta como gema y lo comparte. Instalar una gema toma segundos y te ahorra días. En esta unidad vas a aprender a buscarlas, instalarlas, cargarlas con require y administrarlas con Bundler y un Gemfile, que es como una lista de compras oficial de tu proyecto. Sin Bundler, los proyectos se vuelven un caos de versiones; con Bundler, cualquiera puede clonar tu proyecto y hacerlo funcionar con un solo comando.

La segunda puerta es la depuración y las pruebas. Todos los programadores, sin excepción, escriben errores. La diferencia entre un novato y un profesional no es que el profesional no se equivoque, sino que detecta y corrige sus errores mucho más rápido. Vas a aprender las técnicas de depuración más usadas en el mundo real: puts bien colocado, la diferencia entre puts, p y pp, y cómo usar irb como un laboratorio para experimentar antes de escribir código definitivo. También vas a aprender a leer los mensajes de error de Ruby, que al principio parecen jeroglíficos pero en realidad te dicen exactamente dónde está el problema. Y después de depurar, viene el siguiente nivel: las pruebas automatizadas con minitest. Una prueba es un pequeño programa que verifica que tu código hace lo que promete. Cuando tienes pruebas, puedes cambiar tu código con confianza, porque si rompes algo, las pruebas te avisan en segundos.

La tercera puerta son las expresiones regulares, el regex. Son un mini-lenguaje dentro de Ruby para buscar y manipular texto siguiendo patrones. Validar que un email tenga forma de email, extraer todos los teléfonos de un documento, verificar que una contraseña tenga mayúsculas y números: todo eso se hace con unas pocas líneas usando regex. Al principio los regex parecen ruido aleatorio de símbolos, pero una vez que entiendes sus piezas básicas (literales, clases de caracteres, cuantificadores y grupos) se vuelven una herramienta que vas a usar toda tu carrera.

Esta unidad es densa, así que tómala con calma. Primero las gemas con ejemplos reales que vas a instalar en tu máquina. Luego la depuración con casos reales de errores. Después minitest paso a paso. Y al final, regex con ejemplos prácticos que puedes usar mañana mismo en tus programas. Cuando termines, vas a sentir que tu caja de herramientas se triplicó.

Explicación detallada

Qué es una gema y por qué te importan

Imagina que estás armando muebles. Podrías cortar la madera, hacer los tornillos y fabricar el barniz desde cero. O podrías comprar los tornillos hechos, la madera cortada y el barniz enlatado, y concentrarte en el diseño del mueble. Las gemas son eso: piezas ya fabricadas por otros programadores que tú ensamblas en tu proyecto.

Una gema es simplemente un paquete de código Ruby empaquetado con un formato estándar, con su nombre, su versión y una descripción. RubyGems es el sistema que las distribuye, y rubygems.org es el sitio donde viven más de 180,000 gemas. Cuando instalas Ruby, ya viene con varias gemas incluidas.

La analogía más cercana son las apps de tu celular. No programas tu propia app de mapas cuando necesitas llegar a algún lado: instalas la que ya existe. Igual con las gemas: no programas tu propio cliente HTTP cuando necesitas descargar una página, instalas `httparty`. No inventas tu propio generador de datos falsos para probar, instalas `faker`.

ERROR COMÚN DE PRINCIPIANTE: reinventar la rueda. El novato pasa tres días escribiendo su propio sistema para colorear texto en la terminal, con códigos ANSI a mano, lleno de errores. El programador con experiencia instala `colorize` en diez segundos y sigue con lo importante. No es flojera, es inteligencia: tu tiempo vale más que el orgullo de haberlo hecho todo tú.

MALO (reinventar la rueda):

```
def rojo(texto)
  "\e[31m#{texto}\e[0m"
end

def verde(texto)
  "\e[32m#{texto}\e[0m"
end

puts rojo("Error: algo salió mal")
puts verde("Todo bien")
```

Esto funciona, pero es frágil: si te equivocas en un código ANSI, el color se queda pegado y toda tu terminal se pinta. Y solo tienes dos colores.

BUENO (usar la gema):

```
require "colorize"

puts "Error: algo salió mal".red
puts "Todo bien".green
puts "Aviso".yellow.on_blue
```

Diez segundos de instalación, cero errores, decenas de combinaciones. Esa es la lección: antes de programar algo genérico, pregunta si ya existe una gema que lo resuelva.

BUENA PRÁCTICA: cuando empieces un proyecto, dedica quince minutos a buscar gemas para las partes aburridas (HTTP, colores, fechas, datos de prueba, lectura de CSV). Reserva tu energía para la lógica que hace único a tu proyecto.

gem install y gem list: tu primer contacto

Instalar una gema es un comando de terminal. No se hace dentro de tu código Ruby, se hace en la línea de comandos, igual que cuando instalas un programa.

Para instalar la gema colorize:

```
gem install colorize
```

Vas a ver algo como:

```
Fetching colorize-1.1.0.gem
Successfully installed colorize-1.1.0
1 gem installed
```

Fíjate en el número de versión: 1.1.0. Las gemas tienen versiones porque cambian con el tiempo, igual que las apps. Si no especificas versión, se instala la más reciente.

Para ver qué gemas tienes instaladas:

```
gem list
```

Te muestra una lista larga con nombres y versiones. Para buscar una en específico:

```
gem list colorize
```

O para buscar gemas en internet sin salir de la terminal:

```
gem search -r "http client"
```

La -r significa remoto: busca en rubygems.org en vez de en tu máquina.

ERROR COMÚN: escribir `gem install` dentro de `irb` o dentro de un archivo `.rb`. No funciona, porque `gem` es un comando del sistema operativo, no una instrucción de Ruby. Si lo intentas dentro de `irb`, Ruby te va a decir que no conoce ese método. Los comandos `gem` van en la terminal; el código Ruby va en tus archivos.

Otro error típico en Windows o en sistemas con permisos restringidos: que la instalación falle por permisos. La solución moderna es no pelear con eso y usar un manejador de versiones de Ruby como `rbenv` o `rvm`, que instalan todo en tu carpeta de usuario sin necesitar permisos de administrador. Si ves errores de "permission denied" al instalar gemas, ese es el camino.

Para desinstalar una gema que ya no quieres:

```
gem uninstall colorize
```

Y para ver la documentación de una gema instalada:

```
gem help
ri String#upcase
```

El comando `ri` es la documentación integrada de Ruby. Pruébalo, es rapidísimo.

require: cargar la gema en tu programa

Instalar la gema la guarda en tu computadora, pero tu programa todavía no la conoce. Para usarla tienes que cargarla con require al inicio de tu archivo:

```
require "colorize"

puts "Hola".green
```

Sin el require, Ruby no sabe qué es .green y te lanza un NoMethodError. Piensa en require como decirle a Ruby: "oye, trae la caja de herramientas que instalé ayer".

Hay dos formas de require que debes distinguir:

require carga gemas y archivos de la ruta de carga (\$LOAD_PATH). require_relative carga archivos que están junto a tu archivo actual, usando rutas relativas. Para tus propios archivos del proyecto usa require_relative; para gemas usa require.

MALO (confundir los dos):

```
require "../mi_clase"  # funciona pero es frágil
require "colorize"     # bien para gemas
```

BUENO:

```
require "colorize"      # gema instalada
require_relative "mi_clase" # archivo de mi proyecto
```

require_relative es más claro y no depende de desde qué carpeta ejecutes el programa. Acostúmbrate a usarlo para tu propio código.

ERROR COMÚN: olvidar el require y pasar veinte minutos preguntándote por qué el método "no existe". Si instalaste la gema pero Ruby dice uninitialized constant o undefined method, lo primero que debes revisar es si pusiste el require al inicio del archivo. Nueve de cada diez veces, ese es el problema.

Otro detalle: algunas gemas tienen un nombre de gema distinto al nombre que usas en require. Por ejemplo, la gema se llama net-http pero la cargas con require "net/http". Cuando la documentación de la gema diga cómo cargarla, hazle caso a la documentación, no al nombre del paquete.

Gemfile y Bundler: orden en tus proyectos

Cuando tu proyecto usa tres gemas, acordarte de cuáles son es fácil. Cuando usa treinta, es imposible. Y peor: cada gema tiene versiones, y tu programa puede funcionar con la versión 2.1 pero romperse con la 2.2. Si otra persona quiere correr tu proyecto, ¿cómo sabe qué versiones instalar?

Para eso existe el Gemfile: un archivo de texto en la raíz de tu proyecto que declara qué gemas necesita y, si quieres, qué versiones. Es la lista de compras oficial del proyecto.

Un Gemfile típico se ve así:

```
source "https://rubygems.org"

gem "colorize"
gem "faker"
gem "httparty", "~> 0.21"
```

La primera línea dice de dónde descargar las gemas (casi siempre rubygems.org). Cada línea gem declara una dependencia. Lo de "~> 0.21" significa: "cualquier versión 0.21.x, pero no 0.22". Es una forma de decir "quiero actualizaciones pequeñas que corrigen errores, pero no cambios grandes que podrían romper mi código". Este operador ~> se llama "pessimistic version constraint" y lo vas a ver en todos los proyectos Ruby serios.

Bundler es la gema que lee el Gemfile e instala exactamente lo necesario. El flujo de trabajo es:

1. Escribes tu Gemfile.
2. Corres bundle install en la terminal (dentro de la carpeta del proyecto).
3. Bundler instala las gemas y crea un archivo Gemfile.lock con las versiones exactas que instaló.
4. En tu código, en vez de require sueltos, pones al inicio:

```
require "bundler/setup"
Bundler.require
```

Esto carga automáticamente todas las gemas del Gemfile. O más común en scripts pequeños: sigues usando require normal, pero ejecutas tu programa con bundle exec:

```
bundle exec ruby mi_programa.rb
```


bundle exec se asegura de que tu programa use exactamente las versiones del Gemfile.lock, no otras que tengas instaladas en el sistema.

ANALOGÍA: el Gemfile es la receta de cocina ("necesito harina, huevos, leche") y el Gemfile.lock es la foto exacta de los ingredientes que usaste ("harina marca X lote 123"). Con la receta sola, otra persona podría usar harina diferente y el pastel salir distinto. Con la foto, puede replicar tu resultado exacto. Por eso el Gemfile.lock se sube al repositorio en aplicaciones, para que todo el equipo use las mismas versiones.

ERROR COMÚN: instalar gemas con gem install a lo loco en un proyecto con Gemfile y luego preguntarse por qué bundle exec se queja. La regla es simple: si el proyecto tiene Gemfile, todo se instala con bundle install, no con gem install suelto. gem install es para herramientas globales de tu máquina (como la propia bundler); bundle install es para las dependencias de un proyecto.

Para crear un Gemfile rápidamente en un proyecto nuevo:

```
bundle init
```

Te genera un Gemfile básico que luego editas. Después agregas tus gemas y corres bundle install. Así de simple es empezar con orden desde el día uno.

BUENA PRÁCTICA: todo proyecto Ruby que compartas con alguien (o contigo mismo dentro de seis meses) debe tener Gemfile. Es la diferencia entre "clónalo y corre bundle install" y "buena suerte adivinando qué versiones usé".

Gemas famosas que debes conocer: faker, colorize y httparty

Hay miles de gemas, pero algunas son tan útiles que casi todo programador Ruby las conoce. Vamos a ver tres que te van a servir de inmediato.

FAKER: datos falsos para probar. Cuando estás desarrollando, necesitas datos de ejemplo: nombres, correos, direcciones, teléfonos. Llenarlos a mano es tedioso. Faker los genera por ti.

```
gem install faker
```

Y en tu código:

```
require "faker"

5.times do
  puts "#{Faker::Name.name} - #{Faker::Internet.email} -  
#{Faker::PhoneNumber.phone_number}"
end
```

Salida de ejemplo:

```
María González - maria.gonzalez@example.com - 555-123-4567  
Juan Pérez - juan.perez@example.com - 555-987-6543
```

Cada vez que lo corres genera datos distintos. Es perfecto para llenar tu agenda de contactos de prueba, para simular usuarios en tu app, o para no usar datos reales mientras programas. Faker tiene generadores para casi todo: `Faker::Address.city`, `Faker::Lorem.sentence`, `Faker::Number.between`, incluso `Faker::ChuckNorris.fact` si quieres divertirte.

COLORIZE: colores en la terminal. Ya la viste arriba. Además de colores de texto, maneja fondos y estilos:

```
require "colorize"

puts "ERROR".red
puts "Éxito".green
puts "Advertencia".yellow
puts "Info".light_blue
puts "Texto con fondo".black.on_yellow
puts "Negritas".bold
puts "Subrayado".underline
```

Es ideal para que tus programas de consola se vean profesionales: errores en rojo, éxitos en verde, menús resaltados. Tu proyecto final de esta unidad (la agenda de contactos) se va a ver mucho mejor con colorize.

HTTParty: hablar con internet. Hacer peticiones HTTP a mano en Ruby puro es posible pero verboso. HTTParty lo reduce a una línea:

```
gem install httparty
```

```
require "httparty"

respuesta = HTTParty.get("https://api.github.com/users/octocat")
puts respuesta.code
puts respuesta["login"]
puts respuesta["name"]
```

Con eso consultaste la API pública de GitHub y obtuviste los datos del usuario octocat como un hash de Ruby. HTTParty convierte el JSON automáticamente. Así de fácil es conectar tu programa con cualquier servicio web.

ANALOGÍA: estas tres gemas son como los tres primeros utensilios de tu cocina: un buen cuchillo (faker para picar datos de prueba), una sartén antiadherente (colorize para que todo se vea bien) y una licuadora (httparty para mezclarte con internet). Con esos tres ya puedes cocinar muchísimo.

ERROR COMÚN: instalar una gema y usarla sin leer ni un ejemplo de su documentación. Cada gema tiene su página en rubygems.org con ejemplos de uso. Cinco minutos leyendo te ahorran una hora de adivinar. Acostúmbrate: instalar, leer el ejemplo básico, probar en irb, y luego sí usarla en tu proyecto.

Depuración: el arte de encontrar tus errores

Depurar es buscar y corregir errores en tu programa. Todos lo hacemos, todos los días. La diferencia entre sufrir y resolver rápido está en la técnica.

La técnica número uno, la que usan hasta los programadores más experimentados, es el puts debugging: poner puts en puntos estratégicos para ver qué está pasando dentro del programa.

Mira este programa con un error:

```
def promedio(calificaciones)
  suma = 0
  calificaciones.each do |c|
    suma = suma + c
  end
  suma / calificaciones.length
end

puts promedio([8, 9, 10])
```

Si lo corres, imprime 9. Pero el promedio real de 8, 9 y 10 es 9.0... espera, eso está bien. Cambiemos los datos:

```
puts promedio([8, 9, 7])
```

Imprime 8, pero el promedio real es 8.0. Hmm, sigue bien. Prueba con [8, 9, 10, 7]: imprime 8, pero el real es 8.5. ¡Ahí está el error! En Ruby, dividir dos enteros da un entero (se trunca el decimal). $34 / 4 = 8$, no 8.5.

¿Cómo lo encuentras con puts? Así:

```
def promedio(calificaciones)
  suma = 0
  calificaciones.each do |c|
    suma = suma + c
  end
  puts "DEBUG: suma=#{suma}, cantidad=#{calificaciones.length}"
  puts "DEBUG: división=#{suma / calificaciones.length}"
  suma / calificaciones.length
end
```

Al correrlo ves:

```
DEBUG: suma=34, cantidad=4
DEBUG: división=8
```

Ahí lo ves claro: la división de enteros trunca. La solución es convertir a flotante:

BUENO:

```
def promedio(calificaciones)
  suma = 0
  calificaciones.each do |c|
    suma = suma + c
  end
  suma.to_f / calificaciones.length
end

puts promedio([8, 9, 10, 7]) # => 8.5
```

La lección: cuando algo sale mal, no adivines. Imprime los valores intermedios y deja que los datos te digan dónde está el problema. El prefijo "DEBUG:" te ayuda a distinguir tus mensajes

temporales de la salida real del programa, y así los puedes buscar y borrar fácil cuando termines.

puts vs p vs pp: tres formas de mirar tus datos

puts es el que ya conoces: imprime el valor de forma amigable. Pero tiene un detalle: si le pasas un objeto, llama a su método `to_s`, que a veces esconde información.

`p` llama a `inspect` en vez de `to_s`, y te muestra la estructura real del objeto. Mira la diferencia:

```
nombre = "Ana"
numeros = [1, 2, 3]
datos = { nombre: "Ana", edad: 25 }

puts nombre      # => Ana
p nombre         # => "Ana"

puts numeros     # => 1 2 3 (cada uno en su línea)
p numeros        # => [1, 2, 3]

puts datos       # => {:nombre=>"Ana", :edad=>25}
p datos          # => {:nombre=>"Ana", :edad=>25}
```

¿Ves la diferencia clave? Con `puts`, el string "Ana" se imprime sin comillas y el arreglo se desparrama en líneas. Con `p`, ves exactamente qué tipo de dato es: las comillas te dicen "esto es un string", los corchetes te dicen "esto es un arreglo". Cuando estás depurando, esa información vale oro.

El caso clásico: no sabes si una variable es el número 5 o el string "5". Con `puts` ambos se ven igual. Con `p`, uno se ve como 5 y el otro como "5". Misterio resuelto en un segundo.

MALO (depurar a ciegas):

```
puts valor # imprime 5... ¿pero es número o texto? quién sabe
```

BUENO (depurar con información):

```
p valor # imprime "5" -> ¡es un string! por eso la suma fallaba
```

`pp` significa "pretty print" (impresión bonita). Es como `p`, pero formatea estructuras grandes y anidadas de forma legible, con saltos de línea e indentación. Para usarlo necesitas require "pp":

```
require "pp"

usuarios = [
  { nombre: "Ana", edad: 25, cursos: ["ruby", "html"] },
  { nombre: "Luis", edad: 30, cursos: ["css", "js", "python"] }
]

pp usuarios
```

En vez de una línea larguísima, pp te lo muestra ordenado y legible. Cuando depures hashes o arreglos grandes, pp es tu mejor amigo.

REGLA PRÁCTICA: usa puts para la salida normal de tu programa (lo que el usuario debe ver), p para depurar valores (lo que tú necesitas ver), y pp para depurar estructuras grandes. Y cuando termines de depurar, borra o comenta tus líneas DEBUG. Nadie quiere ver "DEBUG: suma=34" en un programa terminado.

irb: tu laboratorio de experimentos

irb (Interactive Ruby) es una consola donde escribes código Ruby y se ejecuta al instante. Ya la mencionamos antes, pero aquí la vas a usar como herramienta de depuración profesional.

Abres irb escribiendo irb en la terminal. Te aparece un prompt como irb(main):001> donde escribes código:

```
irb(main):001> 2 + 2
=> 4
irb(main):002> "hola".upcase
=> "HOLA"
irb(main):003> [1, 2, 3].map { |n| n * 10 }
=> [10, 20, 30]
```

Cada línea se ejecuta en cuanto presionas Enter, y te muestra el resultado con =>. Es como tener un cuaderno de borrador para probar ideas antes de escribirlas en tu programa.

USOS REALES DE IRB EN DEPURACIÓN:

1. Probar un método que no entiendes. ¿No sabes qué hace String#gsub exactamente? Pruébalo en irb con tres ejemplos distintos hasta que lo entiendas, en vez de adivinarlo dentro de tu programa grande.

2. Reproducir un error en pequeño. Si tu programa de 200 líneas falla, copia la parte sospechosa a irb y experimentala aislada. Sin el ruido del resto del programa, el error suele saltar a la vista.
3. Inspeccionar una gema nueva. Instalaste faker y quieres ver qué genera: abre irb, haz require "faker" y juega con Faker::Name.name hasta que lo domines.
4. Verificar una expresión regular. ¿Tu regex para emails funciona? Pruébalo en irb contra cinco ejemplos antes de meterlo a tu código.

Para salir de irb escribe exit o presiona Ctrl+D.

ERROR COMÚN: usar irb como calculadora nada más y nunca como herramienta de depuración. irb es mucho más que una calculadora: es el lugar donde los programadores Ruby prueban hipótesis. "¿Será que el error está aquí?" -> lo pruebas en irb en diez segundos -> confirmas o descartas. Ese ciclo rápido es lo que hace rápido a un programador.

BUENA PRÁCTICA: antes de escribir una función complicada directamente en tu archivo, prototípala en irb. Juega con los datos de ejemplo, ve qué sale, ajusta. Cuando ya funciona en irb, la pasas a tu archivo limpia y probada. Vas a escribir menos errores desde el inicio.

Cómo leer los errores de Ruby sin entrar en pánico

Cuando tu programa falla, Ruby imprime un mensaje de error con un rastreo (backtrace). Al principio parece un muro de texto amenazador. Pero tiene una estructura simple y te dice exactamente qué pasó y dónde.

Mira este ejemplo:

```
mi_programa.rb:5:in `saludar': undefined method `upcase' for nil:NilClass  
(NoMethodError)  
    from mi_programa.rb:10:in `<main>'
```

Vamos a leerlo por partes, de arriba hacia abajo:

1. mi_programa.rb:5 -> el error ocurrió en la línea 5 de tu archivo.
2. in `saludar' -> dentro del método saludar.
3. undefined method `upcase' for nil:NilClass -> intentaste llamar upcase en algo que es nil (nada, vacío).
4. (NoMethodError) -> el tipo de error: llamaste un método que no existe para ese objeto.

5. `from mi_programa.rb:10:in `<main>'` -> la llamada vino de la línea 10, del programa principal.

Traducción a lenguaje humano: "En la línea 5, dentro del método `saludar`, intentaste usar `upcase` en una variable que vale `nil`, o sea que no tiene ningún valor asignado".

LOS ERRORES MÁS COMUNES Y QUÉ SIGNIFICAN:

NoMethodError: `undefined method X for Y`. Llamaste un método que no existe en ese objeto. Causas típicas: error de dedo en el nombre del método (`upacse` en vez de `upcase`), o la variable es `nil` cuando creías que tenía un valor. Solución: usa `p` para ver qué contiene realmente la variable.

NameError: `undefined local variable or method X`. Usaste una variable que no existe (o la escribiste mal). Revisa la ortografía y verifica que la variable se haya creado antes de usarla.

SyntaxError: `unexpected end-of-input`. Te falta cerrar algo: un `end`, una llave, un paréntesis, o una comilla. Ruby llegó al final del archivo esperando que cerraras algo. Cuenta tus `do/end` y tus llaves.

TypeError: `no implicit conversion of String into Integer`. Mezclaste tipos incompatibles, como sumar un string con un número. Revisa con `p` qué tipo es cada valor.

ArgumentError: `wrong number of arguments`. Llamaste un método con más o menos argumentos de los que espera. Revisa la definición del método.

ZeroDivisionError: `divided by 0`. Intentaste dividir entre cero. Verifica que el divisor no sea cero antes de dividir.

LoadError: `cannot load such file`. El `require` no encontró el archivo o la gema. Verifica el nombre, y si es gema, que esté instalada.

ESTRATEGIA PARA LEER ERRORES:

1. Lee la PRIMERA línea del error, no la última. La primera línea tiene el qué y el dónde. El resto es el camino que siguió el programa hasta fallar.
2. Ve a la línea que indica y mira el código.
3. Pregúntate: ¿qué valor tiene cada variable en esa línea? Usa `p` para averiguarlo.
4. Corrige, corre de nuevo, repite.

ANALOGÍA: el mensaje de error es como cuando el doctor te dice "te duele aquí por esta razón". No te enojas con el doctor por decirte dónde duele; agradécele que te ahorró horas de adivinar. Ruby te está ayudando, no regañando.

ERROR COMÚN DE PRINCIPIANTE: ver el error, entrar en pánico, y empezar a cambiar código al azar esperando que funcione. Eso se llama "programación por coincidencia" y casi nunca funciona. El camino profesional es: leer, entender, verificar con `print`, y luego corregir con precisión.

Qué son las pruebas y por qué te van a salvar

Una prueba automatizada es un pequeño programa que verifica que tu código hace lo que debe hacer. La escribes una vez, y la puedes correr mil veces. Cada vez que cambias tu código, corres las pruebas y en segundos sabes si rompiste algo.

Imagina que tienes una función que calcula descuentos en una tienda:

```
def descuento(precio, porcentaje)
  precio = (precio * porcentaje / 100.0)
end
```

Funciona. Pero dentro de seis meses le agregas impuestos, luego redondeo, luego descuentos por volumen. Cada cambio es una oportunidad de romper lo que ya funcionaba. Sin pruebas, la única forma de verificar es probar a mano cada vez: tedioso, lento, y siempre se te olvida algún caso.

Con pruebas, escribes una vez lo que la función debe hacer:

```
assert_equal 90, descuento(100, 10) # 10% de 100 = 90
assert_equal 50, descuento(100, 50) # 50% de 100 = 50
assert_equal 100, descuento(100, 0) # 0% no descuenta nada
```

Y cada vez que cambies la función, corres las pruebas. Si alguna falla, sabes exactamente qué se rompió. Es como tener un asistente que revisa tu trabajo en segundos, sin cansarse nunca.

ANALOGÍA: las pruebas son como los frenos de un carro. No te hacen ir más rápido, pero te permiten ir rápido con confianza. Un programador sin pruebas maneja despacio y con miedo a cada cambio. Un programador con pruebas puede refactorizar (mejorar el código) sin miedo, porque si algo se rompe, se entera en segundos.

"Pero escribir pruebas toma tiempo". Sí, al principio. Pero piensa en el tiempo que pierdes depurando errores que una prueba habría atrapado en segundos. Los profesionales no escriben pruebas porque les sobre tiempo; escriben pruebas porque no les sobra tiempo para depurar lo mismo dos veces.

minitest paso a paso: tu primera prueba

minitest es la herramienta de pruebas que viene incluida con Ruby. No necesitas instalar nada. Es simple y directa, perfecta para empezar.

Vamos a probar una clase Calculadora. Primero, el código a probar (guárdalo como calculadora.rb):

```
class Calculadora
  def sumar(a, b)
    a + b
  end

  def restar(a, b)
    a - b
  end

  def dividir(a, b)
    raise ZeroDivisionError, "No se puede dividir entre cero" if b == 0
    a.to_f / b
  end
end
```

Ahora, el archivo de pruebas (guárdalo como test_calculadora.rb):

```

require "minitest/autorun"
require_relative "calculadora"

class TestCalculadora < Minitest::Test
  def setup
    @calc = Calculadora.new
  end

  def test_suma_numeros_positivos
    assert_equal 5, @calc.sumar(2, 3)
  end

  def test_suma_con_negativos
    assert_equal 1, @calc.sumar(5, -4)
  end

  def test_resta
    assert_equal 7, @calc.restar(10, 3)
  end

  def test_division_normal
    assert_equal 2.5, @calc.dividir(5, 2)
  end

  def test_division_entre_cero_lanza_error
    assert_raises(ZeroDivisionError) do
      @calc.dividir(10, 0)
    end
  end
end

```

Vamos a desmenuzar cada parte:

`require "minitest/autorun"`: carga minitest y hace que las pruebas se ejecuten solas al correr el archivo.

`class TestCalculadora < Minitest::Test`: tu clase de pruebas hereda de `Minitest::Test`. Así minitest sabe que es una suite de pruebas.

`def setup`: este método especial se ejecuta ANTES de cada prueba. Aquí creas los objetos que todas las pruebas necesitan. En este caso, una calculadora fresca para cada prueba, para que no se contaminen entre sí.

def test_*: cada método cuyo nombre empiece con test_ es una prueba. El nombre debe describir qué verifica: test_suma_numeros_positivos se entiende solo.

assert_equal esperado, real: la afirmación más usada. Verifica que el valor real sea igual al esperado. Si no lo es, la prueba falla y te muestra ambos valores.

assert_raises(ErrorEsperado) do ... end: verifica que el bloque lance ese error. Útil para probar que tu código maneja bien los casos malos.

Para correr las pruebas:

```
ruby test_calculadora.rb
```

Salida:

```
Run options: --seed 12345

# Running:

.....

Finished in 0.002s, 2500.0 runs/s, 5 assertions/s.

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Los puntitos son pruebas que pasaron. "5 runs, 5 assertions, 0 failures" significa: se ejecutaron 5 pruebas, 5 afirmaciones, cero fallas. Todo verde.

Ahora mira qué pasa si introduces un error a propósito en la calculadora (cambia sumar para que reste):

```
def sumar(a, b)
  a - b # error a propósito
end
```

Al correr las pruebas:

```
Failure:
TestCalculadora#test_suma_numeros_positivos [test_calculadora.rb:12]:
Expected: 5
Actual: -1
```

minitest te dice exactamente qué prueba falló, en qué línea, qué esperaba y qué obtuvo. En dos segundos localizaste el error. Sin pruebas, habrías tenido que probar a mano hasta darte cuenta.

LOS ASSERTS MÁS ÚTILES:

- `assert_equal` esperado, real: verifica igualdad.
- `assert_not_equal` no_esperado, real: verifica que NO sean iguales.
- `assert` valor: verifica que sea verdadero (truthy).
- `refute` valor: verifica que sea falso (falsy). Es lo opuesto de `assert`.
- `assert_nil` valor: verifica que sea nil.
- `assert_includes` coleccion, elemento: verifica que la colección contenga el elemento.
- `assert_raises(Error) { }`: verifica que se lance el error.
- `assert_match` patron, string: verifica que el string cumpla el patrón (útil con regex, que verás al final de esta unidad).

BUENA PRÁCTICA: prueba los casos normales, los casos borde (cero, vacío, negativos) y los casos de error. La mayoría de los errores reales viven en los bordes, no en el camino feliz. Si tu función divide, prueba dividir entre cero. Si recibe un arreglo, prueba con un arreglo vacío. Esos casos borde son los que te van a dar dolores de cabeza en producción.

ERROR COMÚN: escribir pruebas que no prueban nada, como `assert_equal 5, 5`. La prueba debe ejercitar TU código, no verificar que Ruby sabe sumar. Cada `assert` debe llamar a tu método con datos de entrada y comparar contra el resultado esperado.

Expresiones regulares: el mini-lenguaje para dominar texto

Una expresión regular (regex) es un patrón que describe un conjunto de textos. En Ruby se escribe entre diagonales: `/patrón/`. Sirve para buscar, validar y reemplazar texto de forma precisa.

El ejemplo más simple: buscar una palabra.

```
texto = "Me encanta programar en Ruby"
if texto =~ /Ruby/
  puts "Encontré la palabra Ruby"
end
```

El operador `=~` compara un string contra un regex y devuelve la posición donde encontró la coincidencia (o nil si no hay). También puedes usar el método `match`, que devuelve un objeto `MatchData` con más información:

```
resultado = "Me encanta programar en Ruby".match(/programar/)
puts resultado[0] # => "programar"
```

ANALOGÍA: un regex es como la descripción que le das a alguien para encontrar una casa: "es la casa azul con portón negro, la tercera de la esquina". No le das la dirección exacta, le das un patrón. El regex describe la FORMA del texto que buscas, no el texto exacto. Por eso sirve para validar emails: no sabes cuál email te van a dar, pero sabes qué forma debe tener.

Clases de caracteres: describir qué letras aceptas

En un regex, la mayoría de los caracteres se buscan a sí mismos: `/gato/` busca literalmente "gato". Pero hay caracteres especiales que describen grupos:

- `.` (punto): cualquier carácter excepto salto de línea. `/g.to/` encuentra "gato", "gito", "g3to".
- `\d`: cualquier dígito (0-9). Equivale a `[0-9]`.
- `\D`: cualquier cosa que NO sea dígito.
- `\w`: cualquier carácter de palabra (letras, dígitos, guion bajo). Equivale a `[a-zA-Z0-9_]`.
- `\W`: lo opuesto: lo que no sea carácter de palabra (espacios, signos).
- `\s`: espacio en blanco (espacio, tab, salto de línea).
- `\S`: lo que no sea espacio en blanco.

Y los corchetes `[]` definen tus propios grupos:

- `[aeiou]`: cualquiera de esas vocales.
- `[a-z]`: cualquier letra minúscula de la a a la z.
- `[A-Z]`: cualquier mayúscula.
- `[0-9]`: cualquier dígito.
- `^[aeiou]`: el `^` al inicio NIEGA: cualquier cosa que NO sea vocal.

Ejemplos:

```
"hola123" =~ /\d/      # => 4 (encontró un dígito en la posición 4)
"hola" =~ /\d/         # => nil (no hay dígitos)
"Tel: 5551234" =~ /[0-9]+/ # encuentra los números
"ABC" =~ /^[A-Z]+$/    # => 0 (todo son mayúsculas)
```

Fíjate en ^ y \$: fuera de corchetes, ^ significa "inicio del texto" y \$ significa "fin del texto". /^[A-Z]+\$/ significa: "desde el inicio hasta el fin, solo mayúsculas". Son las anclas que convierten una búsqueda parcial en una validación total. Sin ellas, /[A-Z]+/ encontraría mayúsculas en "hola MUNDO adiós" y diría que sí, aunque el texto tenga minúsculas. Con las anclas, exiges que TODO el texto cumpla el patrón.

ERROR COMÚN: olvidar las anclas al validar. Quieres validar que un código postal tenga 5 dígitos y escribes /\d{5}/. El texto "abc12345xyz" pasa la validación porque contiene 5 dígitos en medio. Lo correcto es /\d{5}\$/: exactamente 5 dígitos, ni más ni menos, de inicio a fin.

Cuantificadores: cuántas veces se repite

Los cuantificadores dicen cuántas veces debe aparecer lo anterior:

- *: cero o más veces. /ab*c/ encuentra "ac", "abc", "abbbc".
- +: una o más veces. /ab+c/ encuentra "abc", "abbbc", pero NO "ac".
- ?: cero o una vez (opcional). /colou?r/ encuentra "color" y "colour".
- {n}: exactamente n veces. /\d{5}/ encuentra exactamente 5 dígitos.
- {n,}: n o más veces. /\d{2,}/ encuentra 2 o más dígitos.
- {n,m}: entre n y m veces. /\d{2,4}/ encuentra de 2 a 4 dígitos.

Ejemplos prácticos:

```
# Un número de teléfono simple de 10 dígitos
"5551234567" =~ /\d{10}$/ # => 0 (válido)

/https?:\/\/\// =~ "http://" # el ? hace la "s" opcional: http o https
```

Fíjate en ese último: para buscar una diagonal literal dentro del regex hay que escapara con \ porque la diagonal delimita el regex. Los caracteres especiales (. * + ? [] () { } ^ \$ | \) se escapan con backslash cuando los quieres literales.

MALO (validar teléfono sin pensar):

```
/\d+/ # encuentra dígitos en cualquier parte, no valida nada
```

BUENO (validar formato específico):

```
/^\d{3}-\d{3}-\d{4}$/ # formato 555-123-4567, exacto
```

Grupos y captura: extraer partes del texto

Los paréntesis () agrupan partes del patrón Y capturan lo que encontraron, para que lo puedas usar después. Es la parte más poderosa del regex.

```
texto = "Mi teléfono es 555-123-4567"
resultado = texto.match(/(\d{3})-(\d{3})-(\d{4})/)

puts resultado[0] # => "555-123-4567" (todo lo que coincidió)
puts resultado[1] # => "555" (primer grupo)
puts resultado[2] # => "123" (segundo grupo)
puts resultado[3] # => "4567" (tercer grupo)
```

Cada par de paréntesis crea un grupo numerado. El grupo 0 es toda la coincidencia; los grupos 1, 2, 3 son cada paréntesis en orden.

Esto sirve para extraer datos estructurados de texto desordenado. Ejemplo: sacar fecha de un texto:

```
texto = "La reunión es el 2026-09-25 a las 10:00"
m = texto.match(/(\d{4})-(\d{2})-(\d{2})/)
puts "Año: #{m[1]}, Mes: #{m[2]}, Día: #{m[3]}"
# => Año: 2026, Mes: 09, Día: 25
```

También puedes usar los grupos dentro del mismo regex con \1, \2 (backreferences), o reemplazar con gsub usando \1:

```
# Quitar etiquetas HTML simples
"<b>hola</b>".gsub(/<\/?b>/, "") # => "hola"

# Reordenar fecha de DD/MM/YYYY a YYYY-MM-DD
"25/09/2026".gsub(/(\d{2})\/(\d{2})\/(\d{4})/, '\3-\2-\1')
# => "2026-09-25"
```


En el reemplazo, \1, \2, \3 se refieren a lo que capturó cada grupo. Así reordenas texto con una sola línea.

Otro uso: el operador | (o lógico) dentro de grupos:

```
/(gato|perro|pájaro)/ # encuentra cualquiera de esas palabras
```

ERROR COMÚN: usar paréntesis solo para agrupar cuando no necesitas capturar, y luego confundirte con los números de grupo. Si solo quieres agrupar sin capturar, usa (?:...):

```
/(?:http|https):\/\/\// # agrupa sin crear grupo de captura
```

Así tus grupos numerados quedan limpios y solo capturan lo que realmente vas a usar.

Ejemplos reales: validar email y extraer teléfonos

Vamos a lo práctico. Validar un email perfectamente según el estándar oficial requiere un regex monstruoso. Pero para uso real, un regex razonable atrapa el 99% de los casos:

```
def email_valido?(email)
  email =~ /\A[\w+\-\.]+\@[a-z\d\-\.\.]+\.[a-z]+\z/i ? true : false
end

puts email_valido?("ana@example.com")      # => true
puts email_valido?("juan.perez@empresa.mx") # => true
puts email_valido?("mal@@example.com")     # => false
puts email_valido?("sin-arroba.com")       # => false
puts email_valido?("ana@example")          # => false
```

Desmenucemos el patrón `\A[\w+\-\.]+\@[a-z\d\-\.\.]+\.[a-z]+\z/i`:

- `\A`: inicio del string (más estricto que `^`, que en Ruby puede coincidir con inicio de línea en textos multilínea).
- `[\w+\-\.]+`: uno o más caracteres de palabra, +, - o punto (la parte del usuario).
- `@`: la arroba literal.
- `[a-z\d\-\.\.]+`: el dominio: letras, dígitos, guion, punto.
- `\.`: un punto literal (escapado porque `.` solo significa "cualquier carácter").
- `[a-z]+`: la extensión (.com, .mx, .org).
- `\z`: fin del string (más estricto que `$`).

- i al final: insensible a mayúsculas/minúsculas.

BUENA PRÁCTICA: en Ruby prefiere \A y \z sobre ^ y \$ para validaciones, porque ^ y \$ coinciden con inicios/fines de LÍNEA, y un texto multilínea malicioso podría colar contenido. Es un detalle de seguridad que los profesionales conocen.

Extraer todos los teléfonos de un texto con scan:

```
texto = "Llama al 555-123-4567 o al 555-987-6543, también al 5551112233"
telefonos = texto.scan(/\d{3}-?\d{3}-?\d{4}/)
p telefonos
# => ["555-123-4567", "555-987-6543", "5551112233"]
```

scan encuentra TODAS las coincidencias y las devuelve en un arreglo. El -? hace el guion opcional, así encuentra teléfonos con y sin guiones. Con una línea extrajiste todos los teléfonos de un documento. Imagina hacer eso a mano con index y substring: serían veinte líneas llenas de errores.

Otro ejemplo real: limpiar y normalizar datos:

```
# Extraer el usuario y dominio de un email
"ana.perez@empresa.com".match(/([^\@]+)@(.+)/)
# [1] => "ana.perez", [2] => "empresa.com"

# Verificar que una contraseña sea segura: mínimo 8 caracteres,
# al menos una mayúscula, una minúscula y un dígito
def password_segura?(pass)
  pass =~ /\A(?=.*[a-z])(?=.*[A-Z])(?=.*\d){8,}\z/ ? true : false
end

puts password_segura?("abc123")      # => false (sin mayúscula, muy corto)
puts password_segura?("Secreto123")  # => true
```

Ese (?=...) se llama lookahead: verifica que algo exista más adelante sin consumirlo. Es un truco avanzado pero muy usado en validaciones de contraseñas. No te preocupes si no lo dominas hoy; con saber que existe ya vas adelante.

ERROR COMÚN: intentar resolver TODO con un solo regex gigante e ilegible. Si tu regex tiene más de 60 caracteres y ni tú lo entiendes una semana después, divídelo en pasos o usa varios regex simples. Un código claro con tres regex pequeños le gana a un regex genio que nadie puede mantener.

Versiones semánticas: el idioma de los números de versión

Cuando viste `colorize-1.1.0` te preguntaste qué significan esos números. Casi todas las gemas usan versionado semántico: MAJOR.MINOR.PATCH.

- MAJOR (el primer número): cambios grandes que pueden romper tu código. Si una gema pasa de 1.x a 2.0, revisa qué cambió antes de actualizar.
- MINOR (el segundo): funciones nuevas, pero sin romper lo existente. Actualizar de 1.1 a 1.2 debería ser seguro.
- PATCH (el tercero): corrección de errores. De 1.1.0 a 1.1.1 solo se arreglaron bugs.

Por eso en el Gemfile se usa `~> 0.21`: significa "acepto 0.21.0, 0.21.1, 0.21.99... pero NO 0.22". Te da las correcciones de errores automáticamente, sin arriesgarte a cambios mayores. Es el equilibrio entre estar actualizado y no romper tu proyecto.

Cuando `bundle install` resuelve las versiones, escribe el resultado exacto en `Gemfile.lock`. Si abres ese archivo verás algo como:

```
GEM
  specs:
    colorize (1.1.0)
    faker (3.2.1)
```

Esas son las versiones exactas instaladas. Si tu compañero clona el proyecto y corre `bundle install`, obtiene exactamente las mismas. Sin el lock, él podría obtener `faker 3.5` con cambios que rompan tu código, y pasarían horas averiguando por qué "en mi máquina sí funciona". El `Gemfile.lock` elimina esa frase de una vez por todas.

BUENA PRÁCTICA: nunca edites el `Gemfile.lock` a mano. Es un archivo generado. Si necesitas cambiar una versión, edita el `Gemfile` y corre `bundle update nombre_gema`. Bundler recalcula y actualiza el lock por ti.

Depuración avanzada: diseccionando un backtrace real

Vamos a practicar con un error real y un poco más enredado, como los que verás en tus proyectos. Guarda esto como `tienda.rb`:

```

class Producto
  attr_reader :nombre, :precio

  def initialize(nombre, precio)
    @nombre = nombre
    @precio = precio
  end

  def con_descuento(porcentaje)
    @precio - (@precio * porcentaje / 100)
  end
end

class Tienda
  def initialize
    @productos = []
  end

  def agregar(producto)
    @productos << producto
  end

  def total_con_descuento(porcentaje)
    @productos.sum { |p| p.con_descuento(porcentaje) }
  end
end

tienda = Tienda.new
tienda.agregar(Producto.new("Laptop", 15000))
tienda.agregar(Producto.new("Mouse", nil))
puts tienda.total_con_descuento(10)

```

Al correrlo:

```

tienda.rb:11:in `-' : nil can't be coerced into Integer (TypeError)
    from tienda.rb:11:in `con_descuento'
    from tienda.rb:24:in `block in total_con_descuento'
    from tienda.rb:24:in `sum'
    from tienda.rb:24:in `total_con_descuento'
    from tienda.rb:30:in `<main>'

```

Vamos a leerlo como profesionales, de arriba hacia abajo:

1. tienda.rb:11:in `': nil can't be coerced into Integer (TypeError). El error real: en la línea 11, el operador - recibió nil donde esperaba un número. Traducción: "@precio es nil".
2. from tienda.rb:11:in `con_descuento': el error ocurrió dentro del método con_descuento.
3. from tienda.rb:24:in `block in total_con_descuento': ese método fue llamado desde el bloque dentro de total_con_descuento (el sum).
4. Las siguientes líneas suben por la cadena: sum, total_con_descuento, hasta <main> (línea 30, tu puts).

El diagnóstico: el producto "Mouse" se creó con precio nil, y al calcular el descuento, nil - algo explota. La línea 11 es la víctima, pero la causa está en la línea 29: Producto.new("Mouse", nil).

¿Y la solución? Depende de tu criterio de diseño, y aquí hay una lección importante: hay dos escuelas.

Opción A (validar al crear): no permitir productos sin precio desde el inicio.

```
def initialize(nombre, precio)
  raise ArgumentError, "El precio no puede ser nil" if precio.nil?
  @nombre = nombre
  @precio = precio
end
```

Opción B (tolerar nil donde se usa): tratar nil como cero en el cálculo.

```
def con_descuento(porcentaje)
  base = @precio || 0
  base - (base * porcentaje / 100)
end
```

La opción A es mejor en la mayoría de los casos: falla rápido y cerca de la causa, en vez de dejar que el nil viaje por tu programa hasta explotar lejos de donde nació. Este principio se llama "fail fast" (falla rápido): mientras más cerca del origen detectes el dato malo, más fácil es depurar. Un nil que viaja diez métodos antes de explotar es una pesadilla; un ArgumentError en el constructor te dice exactamente qué hiciste mal y dónde.

Regex con bloques: reemplazos que piensan

gsub acepta un bloque en vez de un string de reemplazo, y eso lo vuelve muchísimo más poderoso: puedes calcular el reemplazo con código Ruby.

```
# Censurar emails en un texto, dejando ver solo el dominio
texto = "Escríbeme a ana@example.com o a luis@empresa.mx"
censurado = texto.gsub(/[\w.]+\@([\w.]+)/) { |match| "***@#{$1}" }
puts censurado
# => "Escríbeme a ***@example.com o a ***@empresa.mx"
```

Dentro del bloque, \$1, \$2 contienen los grupos capturados del match actual. Aquí capturamos el dominio y reconstruimos un email censurado. Con un string de reemplazo no podrías hacer la lógica de "***@" tan limpio... bueno, sí podrías con '\1', pero con el bloque puedes hacer CUALQUIER cosa: llamar métodos, consultar una base de datos, formatear.

Otro ejemplo: convertir temperaturas en un texto.

```
texto = "Hoy: 25C, mañana: 30C, ayer: 20C."
convertido = texto.gsub(/(\d+)C/) { |m| "#{$1}C (#{ $1.to_i * 9 / 5 + 32 }F)" }
puts convertido
# => "Hoy: 25C (77F), mañana: 30C (86F), ayer: 20C (68F)."
```

El bloque recibe cada coincidencia, extrae los grados, calcula Fahrenheit y devuelve el reemplazo. Esto es imposible con un reemplazo estático. Cuando el reemplazo necesita CÁLCULO, usa gsub con bloque.

Y un último truco de validación: combinar regex con métodos de Ruby en vez de forzar todo dentro del patrón.

MALO (regex monstruoso que valida fecha real):

```
/\A(19|20)\d\d-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])\z/ # ¿y los 29 de febrero?
¿abril con 31?
```

BUENO (regex simple + lógica Ruby):

```
def fecha_valida?(texto)
  return false unless texto =~ /\A\d{4}-\d{2}-\d{2}\z/
  año, mes, dia = texto.split("-").map(&:to_i)
  Date.valid_date?(año, mes, dia)
rescue ArgumentError
  false
end

require "date"
puts fecha_valida?("2026-09-25") # => true
puts fecha_valida?("2026-02-30") # => false (febrero no tiene 30)
```

El regex verifica la FORMA (cuatro dígitos, guion, dos dígitos, guion, dos dígitos) y Date.valid_date? verifica la LÓGICA (que el día exista en ese mes y año). Cada herramienta hace lo que hace mejor. Forzar toda la lógica calendárica dentro de un regex es posible pero miserable de mantener.

Tu primera gema: empaquetar tu propio código

Hasta ahora has sido consumidor de gemas. Pero ¿qué pasa cuando escribes algo útil que quieres reutilizar en varios proyectos, o compartir con el mundo? Empaquetas tu propia gema. Es más fácil de lo que parece.

La forma moderna y simple es usar bundler para generar el esqueleto:

```
bundle gem mi_utilidad
```

Te hace unas preguntas (¿quieres pruebas? ¿qué licencia?) y crea la estructura:

- lib/mi_utilidad.rb: tu código.
- lib/mi_utilidad/version.rb: el número de versión.
- mi_utilidad.gemspec: la descripción del paquete (nombre, versión, autor, dependencias).
- Gemfile, README.md, y lo básico para pruebas.

El gemspec es el corazón: ahí declaras el nombre, la versión, un resumen, y las gemas de las que dependes. Un gemspec mínimo se ve así:

```
Gem::Specification.new do |spec|
  spec.name          = "mi_utilidad"
  spec.version       = "0.1.0"
  spec.authors       = ["Tu Nombre"]
  spec.summary       = "Utilidades para mis proyectos"
  spec.files         = Dir["lib/**/*.rb"]
  spec.require_paths = ["lib"]
end
```

Para construir el paquete:

```
gem build mi_utilidad.gemspec
```

Eso genera `mi_utilidad-0.1.0.gem`, un archivo instalable. Para usarlo en tus propios proyectos sin publicarlo, lo instalas local:

```
gem install ./mi_utilidad-0.1.0.gem
```

Y si quieres compartirlo con el mundo, lo subes a rubygems.org (necesitas una cuenta gratuita):

```
gem push mi_utilidad-0.1.0.gem
```

No tienes que publicar nada hoy. Pero entiende el ciclo completo: alguien escribe código útil, lo empaqueta con un `gemspec`, lo sube a rubygems.org, y tú lo instalas con `gem install`. Ahora sabes exactamente qué hay detrás de cada gema que usas, y sabes que tú también puedes estar del otro lado. Muchas carreras en Ruby empezaron con alguien publicando una gema pequeña que resolvía un problema concreto.

BUENA PRÁCTICA: antes de publicar, pregúntate si tu gema resuelve un problema real y si su nombre es claro. Las mejores gemas hacen UNA cosa bien, tienen un README con ejemplos, y usan versionado semántico desde el día uno.

Ejemplos paso a paso

Ejemplo 1: Tu primera gema en acción. Instala `colorize` en tu terminal con `gem install colorize`. Luego crea este archivo y córrelo. Vas a ver cómo dos líneas transforman la presentación de tu programa.


```
require "colorize"

puts "=== MI PROGRAMA ===".bold.blue
puts "Todo salió bien".green
puts "Cuidado: revisa esto".yellow
puts "Error grave".red.bold
```

Explicación: la primera línea carga la gema. Después, cada string puede encadenar métodos de color: `.green`, `.red`, `.yellow`, `.blue` para el texto; `.bold` para negritas. Fíjate que se pueden encadenar: `.red.bold` es rojo y en negritas. Este patrón lo vas a usar en tu proyecto final para que el menú de la agenda se vea profesional: títulos en azul, errores en rojo, confirmaciones en verde.

Ejemplo 2: Generando datos de prueba con faker. Instala con gem `install faker`. Este programa crea una lista de 10 usuarios falsos, perfecta para probar tu agenda de contactos o cualquier app sin usar datos reales.

```
require "faker"

puts "Generando 10 contactos de prueba..."
puts "-" * 40

10.times do |i|
  nombre = Faker::Name.name
  email = Faker::Internet.email(name: nombre)
  telefono = Faker::PhoneNumber.phone_number
  ciudad = Faker::Address.city
  puts "#{i + 1}. #{nombre} | #{email} | #{telefono} | #{ciudad}"
end
```

Explicación: `Faker::Name.name` genera un nombre completo aleatorio. `Faker::Internet.email` puede recibir el nombre para generar un email relacionado. `Faker::PhoneNumber.phone_number` genera un teléfono con formato. Cada ejecución produce datos distintos, así que puedes correrlo cien veces y siempre tendrás datos frescos para probar. Cuando tu proyecto final necesite contactos de ejemplo, este es el camino.

Ejemplo 3: Consultando una API real con `httparty`. Instala con gem `install httparty`. Este programa consulta la API pública de GitHub y muestra información del usuario. Es tu primer programa que habla con internet de verdad.

```
require "httparty"

usuario = "octocat"
respuesta = HTTParty.get("https://api.github.com/users/#{usuario}")

if respuesta.code == 200
  puts "Usuario: #{respuesta["login"]}"
  puts "Nombre: #{respuesta["name"]}"
  puts "Repositorios públicos: #{respuesta["public_repos"]}"
  puts "Seguidores: #{respuesta["followers"]}"
else
  puts "Error: código #{respuesta.code}"
end
```

Explicación: HTTParty.get hace la petición y devuelve un objeto respuesta. respuesta.code es el código HTTP (200 significa éxito). Como la API devuelve JSON, HTTParty lo convierte automáticamente en un hash de Ruby y accedes a los datos con respuesta["login"]. Cambia "octocat" por cualquier usuario de GitHub y funciona. Así de simple es consumir APIs: esto es la base de miles de aplicaciones reales.

Ejemplo 4: Depuración con p para cazar un error de tipos. Este programa falla de forma misteriosa. Usa p para descubrir por qué.

```
# Programa con error: suma precios pero algo sale mal
precios = ["100", "250", "75"]

total = 0
precios.each do |precio|
  total = total + precio
end

puts "Total: #{total}"
```

Si lo corres, obtienes un TypeError: no implicit conversion of String into Integer. El error te dice la línea, pero para ENTENDERLO agregas p:

```

precios = ["100", "250", "75"]
p precios # => ["100", "250", "75"] ¡son strings, no números!

total = 0
precios.each do |precio|
  p precio # => "100" (con comillas: es texto)
  total = total + precio.to_i # convertimos a entero
end

puts "Total: #{total}" # => Total: 425

```

Explicación: p reveló que los precios eran strings (se ven las comillas). Sumar un string a un entero es imposible en Ruby, de ahí el `TypeError`. La solución es convertir con `to_i` antes de sumar. Este es el error más común cuando lees datos de archivos o del usuario: todo lo que entra por gets o se lee de un archivo llega como string, y hay que convertirlo. Con p lo detectas en segundos.

Ejemplo 5: irb como laboratorio antes de programar. Imagina que necesitas una función que censure palabras prohibidas en un texto. En vez de escribirla directo en tu archivo, la prototipas en irb. Esto es lo que escribirías en tu sesión de irb:

```

irb(main):001> texto = "Este producto es malo y horrible"
=> "Este producto es malo y horrible"
irb(main):002> prohibidas = ["malo", "horrible"]
=> ["malo", "horrible"]
irb(main):003> resultado = texto.dup
=> "Este producto es malo y horrible"
irb(main):004> prohibidas.each { |p| resultado.gsub!(p, "****") }
=> ["malo", "horrible"]
irb(main):005> resultado
=> "Este producto es *** y ****"

```

Funciona. Ahora sí la pasas a tu archivo, limpia y probada:

```

def censurar(texto, prohibidas)
  resultado = texto.dup
  prohibidas.each { |palabra| resultado.gsub!(palabra, "****") }
  resultado
end

puts censurar("Este producto es malo y horrible", ["malo", "horrible"])
# => "Este producto es *** y ****"

```

Explicación: en irb probaste la idea con datos reales en segundos, sin crear archivos ni correr programas completos. Cuando la lógica funcionó, la empaquetaste en un método. Este flujo (prototipar en irb, luego formalizar) te ahorra muchísimos ciclos de "escribir, correr, fallar, corregir".

Ejemplo 6: Tu primera suite de pruebas con minitest. Guarda la clase en validador.rb y las pruebas en test_validador.rb. Luego corre ruby test_validador.rb.

```
# validador.rb
class Validador
  def email_valido?(email)
    email =~ /\A[\w+\-\.]+\@[a-z\d\-\.\+\.][a-z]+\z/i ? true : false
  end

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

```

# test_validador.rb
require "minitest/autorun"
require_relative "validador"

class TestValidador < Minitest::Test
  def setup
    @v = Validador.new
  end

  def test_email_correcto
    assert_equal true, @v.email_valido?("ana@example.com")
  end

  def test_email_sin_arroba
    assert_equal false, @v.email_valido?("anaexample.com")
  end

  def test_email_con_doble_arroba
    assert_equal false, @v.email_valido?("ana@@example.com")
  end

  def test_mayor_de_edad_limite
    assert_equal true, @v.mayor_de_edad?(18)
    assert_equal false, @v.mayor_de_edad?(17)
  end
end

```

Explicación: aquí combinas dos temas de la unidad: el validador usa el regex de emails que aprendiste, y las pruebas verifican que funcione. Fíjate en `test_mayor_de_edad_limite`: prueba el caso borde (18 vs 17), que es donde suelen esconderse los errores. Corre las pruebas y verás "4 runs, 5 assertions, 0 failures". Ahora cambia el regex a propósito para que falle y mira cómo minitest te lo reporta. Esa es la red de seguridad en acción.

Ejemplo 7: Extractor de datos con regex y scan. Este programa lee un texto largo y extrae todos los emails y teléfonos que encuentre. Es el tipo de programa que en el mundo real te piden en el trabajo.

```

texto = <<~TEXT0
  Contacto: María López, maria.lopez@empresa.com, tel 555-123-4567.
  Soporte: ayuda@empresa.com o llama al 5559876543.
  Ventas: ventas@empresa.com, 555-111-2233.
TEXT0

emails = texto.scan(/[\w+\-\.]+\@[a-z\d\-\.]+\.[a-z]+/i)
telefonos = texto.scan(/\d{3}-?\d{3}-?\d{4}/)

puts "Emails encontrados:"
emails.each { |e| puts "  - #{e}" }

puts "Teléfonos encontrados:"
telefonos.each { |t| puts "  - #{t}" }

```

Explicación: scan recorre todo el texto y devuelve un arreglo con todas las coincidencias. El heredoc <<~TEXT0 permite escribir texto multilínea cómodamente. Con dos líneas de regex extrajiste información estructurada de texto desordenado. Nota que aquí no usamos \A y \z porque NO estamos validando un string completo, estamos BUSCANDO dentro de un texto largo: contextos distintos, anclas distintas.

Ejemplo 8: Reemplazo inteligente con grupos de captura. Este programa normaliza fechas: las encuentra en formato DD/MM/YYYY dentro de un texto y las reescribe como YYYY-MM-DD.

```

texto = "Nací el 25/09/1990 y mi hermana el 03/01/1995."

normalizado = texto.gsub(/(\d{2})\/(\d{2})\/(\d{4})/, '\3-\2-\1')
puts normalizado
# => "Nací el 1990-09-25 y mi hermana el 1995-01-03."

```

Explicación: el regex tiene tres grupos: (\d{2}) captura el día, (\d{2}) el mes, (\d{4}) el año. En el reemplazo, '\3-\2-\1' reordena: año-mes-día. gsub lo aplica a TODAS las fechas del texto de una sola pasada. Fíjate en las comillas simples del reemplazo: son importantes, porque con comillas dobles Ruby intentaría interpolar \3 como variable. Este patrón (buscar con grupos, reordenar en el reemplazo) lo vas a usar para limpiar datos toda tu vida.

Práctica

1. Instala la gema faker y escribe un programa que genere 5 productos falsos para una tienda: cada producto debe tener nombre (Faker::Commerce.product_name), precio (Faker::Commerce.price) y departamento (Faker::Commerce.department). Muéstralos numerados.

SOLUCIÓN EXPLICADA: primero instalas con `gem install faker` en la terminal. Luego el programa carga la gema y usa los generadores de comercio. `Faker::Commerce` está pensado justo para esto: nombres de productos realistas, precios con decimales y departamentos de tienda.

```
require "faker"

5.times do |i|
  nombre = Faker::Commerce.product_name
  precio = Faker::Commerce.price
  depto = Faker::Commerce.department
  puts "#{i + 1}. #{nombre} - ${precio} (#{depto})"
end
```

Si te marca error de gema no encontrada, verifica dos cosas: que la instalaste (`gem list faker`) y que pusiste el `require` al inicio. El 90% de los errores con gemas son una de esas dos.

2. Escribe un programa que use `colorize` para mostrar un menú: el título en azul y negritas, las opciones en blanco normal, y al final un mensaje de despedida en verde. Luego simula un error y muéstralo en rojo.

SOLUCIÓN EXPLICADA: `colorize` agrega métodos a los strings. Los encadenas según lo que necesites: `.blue` para color, `.bold` para negritas. El orden del encadenamiento no importa.

```
require "colorize"

puts "=== MENÚ PRINCIPAL ===".blue.bold
puts "1. Agregar contacto"
puts "2. Buscar contacto"
puts "3. Salir"
puts "¡Hasta luego!".green
puts "Error: opción inválida".red.bold
```

Practica combinar: `.yellow.on_blue` es texto amarillo con fondo azul, `.underline` subraya. Usa los colores con propósito: rojo para errores, verde para éxito, amarillo para advertencias. Si todo es de colores, nada destaca.

3. Crea un archivo Gemfile para un proyecto que use las gemas faker, colorize y httparty. Luego escribe los comandos que ejecutarías para instalar todo y para correr tu programa usando las versiones del Gemfile.

SOLUCIÓN EXPLICADA: el Gemfile declara las dependencias y Bundler las instala. Los comandos van en la terminal, dentro de la carpeta del proyecto.

```
# Gemfile
source "https://rubygems.org"

gem "faker"
gem "colorize"
gem "httparty"
```

Comandos en la terminal:

```
bundle install
bundle exec ruby mi_programa.rb
```

bundle install lee el Gemfile, instala lo necesario y genera el Gemfile.lock con las versiones exactas. bundle exec ruby mi_programa.rb corre tu programa garantizando que use esas versiones exactas. Si alguien más clona tu proyecto, con solo bundle install tendrá todo igual que tú.

4. El siguiente programa tiene un error. Usa la técnica de puts debugging con p para encontrarlo y corrígelo.

```
edades = ["25", "30", "20"]
suma = 0
edades.each { |e| suma += e }
puts "Promedio: #{suma / edades.length}"
```

SOLUCIÓN EXPLICADA: el error es TypeError porque e es un string. Lo descubres imprimiendo con p: p e muestra "25" con comillas, revelando que es texto. Además hay un segundo error escondido: aunque conviertas a entero, suma / edades.length haría división entera. La corrección completa:


```

edades = ["25", "30", "20"]
suma = 0
edades.each do |e|
  p e          # DEBUG: => "25" (¡es string!)
  suma += e.to_i
end
puts "Promedio: #{suma.to_f / edades.length}" # => 25.0

```

Moraleja: cuando leas datos del usuario o de archivos, casi siempre llegan como strings. Convierte con `to_i` o `to_f` antes de operar. Y usa `p`, no `puts`, para depurar: `puts` te habría mostrado 25 sin comillas y no habrías visto el problema.

5. Escribe pruebas minitest para una clase `Rectangulo` con métodos `area(base, altura)` y `perimetro(base, altura)`. Incluye al menos 4 pruebas: casos normales, un caso con cero y un caso con decimales.

SOLUCIÓN EXPLICADA: primero la clase, luego las pruebas. El `setup` crea el objeto una vez. Cada `test_` verifica un comportamiento. Incluimos el caso borde (cero) porque es donde suelen fallar las cosas.

```

# rectangulo.rb
class Rectangulo
  def area(base, altura)
    base * altura
  end

  def perimetro(base, altura)
    2 * (base + altura)
  end
end

```

```
# test_rectangulo.rb
require "minitest/autorun"
require_relative "rectangulo"

class TestRectangulo < Minitest::Test
  def setup
    @r = Rectangulo.new
  end

  def test_area_normal
    assert_equal 20, @r.area(4, 5)
  end

  def test_area_con_cero
    assert_equal 0, @r.area(0, 5)
  end

  def test_area_con_decimales
    assert_equal 12.5, @r.area(2.5, 5)
  end

  def test_perimetro
    assert_equal 18, @r.perimetro(4, 5)
  end

  def test_perimetro_cuadrado
    assert_equal 16, @r.perimetro(4, 4)
  end
end
```

Corre con ruby test_rectangulo.rb y debes ver "5 runs, 5 assertions, 0 failures". Acostúmbrate a probar el cero y los decimales: son los casos que rompen programas en producción.

6. Escribe un regex que valide códigos postales mexicanos: exactamente 5 dígitos. Pruébalo con estos casos: "06600" (válido), "6600" (inválido, solo 4), "066000" (inválido, 6 dígitos), "abcde" (inválido), "C.P. 06600" (inválido, tiene letras). Explica por qué necesitas las anclas.

SOLUCIÓN EXPLICADA: sin anclas, `/\d{5}/` encontraría 5 dígitos dentro de "C.P. 06600" y diría que es válido. Las anclas `^` y `$` exigen que TODO el string sea exactamente 5 dígitos.

```
def cp_valido?(cp)
  cp =~ /\A\d{5}\z/ ? true : false
end

puts cp_valido?("06600")      # => true
puts cp_valido?("6600")      # => false
puts cp_valido?("066000")    # => false
puts cp_valido?("abcde")     # => false
puts cp_valido?("C.P. 06600") # => false
```

\d{5} significa exactamente 5 dígitos. \A es inicio del string y \z es fin del string. Juntos significan: "el string completo, de inicio a fin, son 5 dígitos y nada más". Usamos \A y \z en vez de ^ y \$ porque en Ruby ^ y \$ coinciden con inicios de línea, y un string multilínea podría engañar la validación.

7. Escribe un programa que extraiga todos los precios (en formato \$123.45) de un texto usando scan con grupos de captura, y que al final imprima la suma total de los precios encontrados.

SOLUCIÓN EXPLICADA: el regex `/\$(\d+)\.(\d{2})/` captura pesos y centavos por separado. scan con grupos devuelve un arreglo de arreglos: `[["123", "45"], ...]`. Convertimos cada parte a número y sumamos.

```
texto = "El pan cuesta $25.50, la leche $30.00 y el queso $120.75."

precios = texto.scan(/\$(\d+)\.(\d{2})/)
p precios # => [["25", "50"], ["30", "00"], ["120", "75"]]

total = precios.sum { |pesos, centavos| "#{pesos}.#{centavos}".to_f }
puts "Total: ${total}" # => Total: $176.25
```

Fíjate en el destructuring del bloque: `|pesos, centavos|` desempaqueta cada par automáticamente. El `\$` escapa el signo de pesos porque `$` solo significa "fin de línea" en regex. Este es un patrón muy común: extraer datos con scan y procesarlos con enumerables, dos temas de este libro trabajando juntos.

8. Escribe un método `normalizar_telefono` que reciba un teléfono en cualquier formato (con guiones, espacios, paréntesis o sin nada) y lo devuelva en formato XXX-XXX-XXXX. Usa `gsub` con regex para quitar todo lo que no sea dígito, valida que queden 10 dígitos, y luego usa grupos de captura para darle formato. Incluye pruebas minitest.

SOLUCIÓN EXPLICADA: primero limpiamos con `gsub(/\D/, "")` que elimina todo lo que no sea dígito. Luego validamos la longitud. Finalmente, capturamos en tres grupos y reordenamos con guiones. Las pruebas cubren varios formatos de entrada.

```
# telefono.rb
def normalizar_telefono(tel)
  digitos = tel.gsub(/\D/, "")
  return nil unless digitos =~ /\A\d{10}\z/
  digitos.gsub(/(\d{3})(\d{3})(\d{4})/, '\1-\2-\3')
end
```

```
# test_telefono.rb
require "minitest/autorun"
require_relative "telefono"

class TestTelefono < Minitest::Test
  def test_con_guiones
    assert_equal "555-123-4567", normalizar_telefono("555-123-4567")
  end

  def test_sin_formato
    assert_equal "555-123-4567", normalizar_telefono("5551234567")
  end

  def test_con_parentesis_y_espacios
    assert_equal "555-123-4567", normalizar_telefono("(555) 123 4567")
  end

  def test_numero_invalido
    assert_nil normalizar_telefono("555-123")
  end
end
```

Este ejercicio junta todo lo de la unidad: regex para limpiar (`\D`), validación con anclas, grupos de captura para formatear, y minitest para verificar. Es exactamente el tipo de función utilitaria que escribirías en un proyecto real.

Resumen de la unidad

- Una gema es un paquete de código reutilizable. Antes de programar algo genérico, busca si ya existe una gema: te ahorra días de trabajo y errores.

- Instala gemas con `gem install nombre` en la terminal, verifica con `gem list`, y cárgalas en tu código con `require "nombre"` al inicio del archivo.
- Usa `require` para gemas y `require_relative` para los archivos de tu propio proyecto. Si olvidas el `require`, Ruby te dirá que el método o la constante no existen.
- El Gemfile declara las dependencias de tu proyecto y Bundler las instala con `bundle install`, generando el Gemfile.lock con las versiones exactas. Corre tu programa con `bundle exec` para usar esas versiones.
- Gemas esenciales: `faker` (datos falsos para probar), `colorize` (colores en la terminal) y `httparty` (peticiones HTTP simples). Lee siempre el ejemplo básico de su documentación antes de usarlas.
- Depura con puts estratégicos (márcalos con `DEBUG:`), usa `p` en vez de `puts` para ver el tipo real de los datos, y `pp` para estructuras grandes y anidadas.
- `irb` es tu laboratorio: prototipa funciones, reproduce errores en pequeño e inspecciona gemas nuevas antes de usarlas en tu programa.
- Lee los errores de arriba hacia abajo: la primera línea dice qué pasó y en qué línea. Los más comunes son `NoMethodError`, `NameError`, `SyntaxError`, `TypeError` y `ArgumentError`.
- Las pruebas automatizadas verifican que tu código hace lo prometido. Con `minitest`: hereda de `Minitest::Test`, usa `setup` para preparar, escribe métodos `test_*` y verifica con `assert_equal`, `assert_raises` y sus compañeros.
- Prueba casos normales, casos borde (cero, vacío, límites) y casos de error. Las pruebas te permiten cambiar código sin miedo.
- Un regex entre `//` describe la forma del texto que buscas. `=~` devuelve la posición o `nil`; `match` devuelve un `MatchData` con detalles; `scan` encuentra todas las coincidencias.
- Clases de caracteres: `\d` dígitos, `\w` palabra, `\s` espacios, `[a-z]` rangos, `[^...]` negación. Cuantificadores: `*` cero o más, `+` uno o más, `?` opcional, `{n}` exacto, `{n,m}` rango.
- Usa anclas `\A` y `\Z` para validaciones estrictas de strings completos. Los paréntesis capturan grupos que recuperas con `[1]`, `[2]` o reordenas con `\1`, `\2` en `gsub`.
- En la próxima unidad vas a encadenar enumerables como un profesional, construir un proyecto completo (una agenda de contactos funcional) y asomarte a Rails, la razón por la que Ruby se hizo famoso en el mundo.

UNIDAD 10: Proyecto final e intro a Rails

Esta es la última unidad del libro, y está diseñada como una celebración de todo lo que aprendiste. Aquí ya no hay conceptos aislados: vas a combinar enumerables avanzados, archivos, clases y buenas prácticas en un proyecto completo y funcional, una agenda de contactos en consola con todas sus operaciones. Y para cerrar, vas a asomarte a Ruby on Rails, el framework que hizo famoso a Ruby en todo el mundo y que sigue dando empleo a miles de programadores.

Empezamos con los enumerables avanzados, las herramientas que separan al que "usa Ruby" del que "piensa en Ruby". Ya conoces `each`, `map` y `select`. Ahora vienen los pesados: `reduce` (también llamado `inject`), que condensa una colección en un solo valor; `group_by`, que agrupa elementos por criterio; `sort_by`, que ordena con elegancia; `flat_map`, que aplana y transforma en un paso; y `lazy`, para trabajar con colecciones infinitas o gigantes sin quedarte sin memoria. Pero más importante que cada método por separado es la idea de encadenarlos como tuberías: los datos entran por un lado, pasan por una serie de transformaciones, y sale el resultado. Esa forma de pensar es la marca de un programador Ruby maduro.

Después viene el proyecto: una agenda de contactos completa en consola. No un ejemplo de juguete de veinte líneas, sino un programa real con menú, con las cuatro operaciones CRUD (crear, leer, actualizar, eliminar... bueno, en nuestro caso agregar, listar, buscar y eliminar), con los datos guardados en un archivo para que no se pierdan al cerrar el programa, y con el código explicado por partes para que entiendas cada decisión de diseño. Cuando termines el proyecto vas a tener algo que mostrar: un programa funcional, hecho por ti, que persiste datos y que un usuario real podría usar.

Y cerramos con Rails. Vas a entender qué es, por qué existe, cómo funciona el patrón MVC (Modelo-Vista-Controlador) con una analogía que no vas a olvidar, y vas a instalarlo, crear tu primera aplicación, levantar el servidor y ver la página de bienvenida con tus propios ojos. Incluso vas a generar tu primer scaffold, que es como magia: Rails escribe por ti un CRUD completo en segundos. No vas a salir de aquí como experto en Rails (eso es otro libro completo), pero sí vas a salir sabiendo qué es, cómo se ve, y con la puerta abierta para seguir.

Tómate esta unidad como tu examen final práctico. Si puedes construir la agenda y levantar tu primera app Rails, estás listo para llamarte programador Ruby.

Explicación detallada

reduce/inject: condensar una colección en un solo valor

Ya sabes que map transforma cada elemento y select filtra. Pero ¿qué haces cuando quieres COMBINAR todos los elementos en un solo resultado? Sumar todos los números, concatenar todos los strings, encontrar el producto total. Para eso está reduce, que también se puede llamar inject (son el mismo método, dos nombres; reduce es el más moderno).

Mira cómo sumarías una lista a la vieja usanza:

```
numeros = [1, 2, 3, 4, 5]
total = 0
numeros.each do |n|
  total = total + n
end
puts total # => 15
```

Funciona, pero necesitas una variable externa (total) que vas modificando. Con reduce:

```
numeros = [1, 2, 3, 4, 5]
total = numeros.reduce(0) { |acumulador, n| acumulador + n }
puts total # => 15
```

¿Cómo funciona? reduce toma un valor inicial (0) y un bloque con dos parámetros: el acumulador (lo que llevas juntado hasta ahora) y el elemento actual. En cada vuelta, el bloque combina ambos y el resultado se convierte en el nuevo acumulador. Al final, te devuelve el acumulador final.

Vamos paso a paso con [1, 2, 3]:

- Inicio: acumulador = 0
- Vuelta 1: acumulador (0) + 1 = 1 -> nuevo acumulador: 1
- Vuelta 2: acumulador (1) + 2 = 3 -> nuevo acumulador: 3
- Vuelta 3: acumulador (3) + 3 = 6 -> nuevo acumulador: 6
- Resultado: 6

ANALOGÍA: reduce es como hacer una bola de nieve. Empiezas con una bolita (el valor inicial), la ruedas por la nieve (la colección), y en cada vuelta se le pega más nieve (el bloque combina). Al final tienes una bolota: un solo objeto formado por todos los elementos.

Si no pasas valor inicial, reduce usa el primer elemento como acumulador inicial y empieza desde el segundo:

```
[1, 2, 3, 4, 5].reduce { |acc, n| acc + n } # => 15
```

Pero cuidado: si la colección está vacía y no diste valor inicial, reduce devuelve nil. Por eso es buena práctica dar siempre el valor inicial explícito, sobre todo con sumas (0) o concatenaciones ("").

reduce no es solo para sumar. Mira estos usos:

```
# Multiplicar todos (factorial de 5)
(1..5).reduce(1) { |acc, n| acc * n } # => 120

# Concatenar palabras en una frase
["hola", "mundo", "ruby"].reduce("") { |acc, p| acc + p + " " }.strip
# => "hola mundo ruby"

# Contar cuántos aprobaron (calificación >= 6)
calificaciones = [8, 5, 9, 4, 7, 10]
aprobados = calificaciones.reduce(0) { |acc, c| c >= 6 ? acc + 1 : acc }
# => 4

# Encontrar la palabra más larga
["sol", "murciélago", "luz"].reduce { |acc, p| p.length > acc.length ? p : acc }
# => "murciélago"

# Agrupar manualmente (antes de conocer group_by)
[1, 2, 3, 4, 5, 6].reduce({ pares: [], impares: [] }) do |acc, n|
  n.even? ? acc[:pares] << n : acc[:impares] << n
  acc
end
# => {:pares=>[2, 4, 6], :impares=>[1, 3, 5]}
```

Fíjate en ese último ejemplo: el acumulador puede ser CUALQUIER cosa, incluso un hash. Y nota el detalle importante: el bloque debe devolver el acumulador al final (acc), porque el valor de retorno del bloque se convierte en el nuevo acumulador. Si olvidas devolverlo, el acumulador se corrompe. Ese es el error más común con reduce.

MALO (olvidar devolver el acumulador):

```
[1, 2, 3].reduce([]) do |acc, n|
  acc << n * 2
  # el bloque devuelve el resultado de <<, que Sí es acc... funciona por accidente
end
```


En este caso funciona por accidente porque << devuelve el arreglo modificado. Pero mira este:

```
[1, 2, 3].reduce(0) do |acc, n|
  puts "sumando #{n}"
  acc + n
  # si aquí hubiera un puts al final, el bloque devolvería nil y todo se rompería
end
```

BUENO (explícito y seguro):

```
[1, 2, 3].reduce(0) do |acc, n|
  nuevo = acc + n
  puts "Llevo #{nuevo}"
  nuevo # última línea: el nuevo acumulador, clarito
end
```

REGLA DE ORO: la última línea del bloque de reduce debe ser el nuevo valor del acumulador. Si pones un puts al final "para depurar", el bloque devuelve nil (lo que devuelve puts) y tu acumulador se vuelve nil en la siguiente vuelta. Depura ANTES de la última línea, no después.

Hay un atajo muy común: cuando el bloque solo llama un operador, puedes usar el símbolo:

```
[1, 2, 3, 4, 5].reduce(0, :+) # => 15 (suma)
[1, 2, 3, 4, 5].reduce(1, :*) # => 120 (multiplica)
["a", "b", "c"].reduce("", :+) # => "abc" (concatena)
```

.reduce(0, :+) se lee: "reduce empezando en 0, aplicando + entre el acumulador y cada elemento". Es la forma idiomática de sumar en Ruby moderno. De hecho, para sumas simples existe sum:

```
[1, 2, 3, 4, 5].sum # => 15
```

Pero reduce es más general: úsalo cuando la combinación no sea una simple suma.

group_by: organizar datos por categorías

group_by toma una colección y la organiza en grupos según el criterio que le des en el bloque. Devuelve un hash donde las llaves son los criterios y los valores son arreglos con los elementos de cada grupo.

```

numeros = [1, 2, 3, 4, 5, 6]
grupos = numeros.group_by { |n| n.even? ? "par" : "impar" }
p grupos
# => {"par"=>[2, 4, 6], "impar"=>[1, 3, 5]}

```

El bloque se ejecuta para cada elemento y lo que devuelva se usa como llave del grupo. Los elementos que devuelvan lo mismo caen en el mismo grupo.

ANALOGÍA: group_by es como el cartero que separa cartas por colonia. No le importa el contenido de cada carta, solo lee el código postal (el criterio del bloque) y la echa en la caja correspondiente. Al final tienes una caja por colonia, cada una con sus cartas.

Ejemplos del mundo real:

```

# Agrupar palabras por su primera letra
palabras = ["gato", "perro", "gallina", "pato", "pájaro"]
palabras.group_by { |p| p[0] }
# => {"g"=>["gato", "gallina"], "p"=>["perro", "pato", "pájaro"]}

# Agrupar estudiantes por si aprobaron
estudiantes = [
  { nombre: "Ana", calif: 9 },
  { nombre: "Luis", calif: 5 },
  { nombre: "María", calif: 8 },
  { nombre: "Pedro", calif: 4 }
]
estudiantes.group_by { |e| e[:calif] >= 6 ? "aprobados" : "reprobados" }
# => {"aprobados"=>[{"...Ana..."}, {"...María..."}], "reprobados"=>[{"...Luis..."}, {"...Pedro..."}]}

# Agrupar ventas por mes
ventas = [
  { mes: "enero", monto: 100 },
  { mes: "febrero", monto: 200 },
  { mes: "enero", monto: 150 }
]
por_mes = ventas.group_by { |v| v[:mes] }
# => {"enero"=>[{"...100..."}, {"...150..."}], "febrero"=>[{"...200..."}]}

```

Y lo poderoso es combinarlo con transformaciones. ¿Ventas totales por mes?

```
ventas_por_mes = ventas.group_by { |v| v[:mes] }
totales = ventas_por_mes.transform_values { |lista| lista.sum { |v| v[:monto] } }
p totales # => {"enero"=>250, "febrero"=>200}
```

`transform_values` aplica un bloque a cada valor del hash (cada lista) y devuelve un hash nuevo. `group_by` + `transform_values` es una combinación que vas a usar muchísimo en reportes y análisis de datos.

ERROR COMÚN: esperar que `group_by` devuelva los grupos ordenados o en cierto orden. El hash resultante mantiene el orden de primera aparición de cada llave, no un orden alfabético ni nada parecido. Si necesitas orden, ordénalo después con `sort`.

sort_by: ordenar con elegancia

Ya conoces `sort`, que ordena usando el operador `<=>`. Pero cuando quieres ordenar por un CRITERIO calculado (por longitud, por un campo, por una fórmula), `sort_by` es más claro y más rápido.

```
palabras = ["murciélago", "sol", "elefante", "luz"]
palabras.sort_by { |p| p.length }
# => ["sol", "luz", "elefante", "murciélago"]
```

El bloque calcula la "llave de ordenamiento" de cada elemento, y Ruby ordena por esas llaves. Es más eficiente que `sort` con bloque porque calcula cada llave una sola vez (`sort` con bloque las recalcularía en cada comparación).

Ordenar por múltiples criterios: devuelve un arreglo de llaves.

```
estudiantes = [
  { nombre: "Ana", calif: 9, edad: 20 },
  { nombre: "Luis", calif: 9, edad: 19 },
  { nombre: "María", calif: 8, edad: 22 }
]

# Primero por calificación descendente, luego por edad ascendente
ordenados = estudiantes.sort_by { |e| [-e[:calif], e[:edad]] }
# Luis (9, 19), Ana (9, 20), María (8, 22)
```

El truco del signo menos: para ordenar descendente un número, niega la llave. `[-calif, edad]` ordena por calificación de mayor a menor, y en empates, por edad de menor a mayor. Los arreglos se comparan elemento por elemento, así que esto funciona perfecto.

Para orden descendente simple también puedes encadenar reverse:

```
[3, 1, 2].sort_by { |n| n }.reverse # => [3, 2, 1]
```

O más idiomático para números: sort solito ya ordena ascendente, y para descendente:

```
[3, 1, 2].sort.reverse # => [3, 2, 1]
```

MALO (sort con bloque recalculando):

```
palabras.sort { |a, b| a.length <=> b.length } # funciona pero recalcula length  
muchas veces
```

BUENO (sort_by calculando una vez):

```
palabras.sort_by { |p| p.length } # más claro y más rápido
```

BUENA PRÁCTICA: usa sort para orden natural, sort_by cuando el criterio sea una transformación del elemento. Y recuerda que ninguno modifica el arreglo original: devuelven uno nuevo. Si quieres modificar en el lugar, usa sort! y sort_by! (con signo de admiración).

flat_map: transformar y aplanar en un paso

A veces map te devuelve un arreglo de arreglos y lo que quieres es un solo arreglo plano. Podrías hacer map y luego flatten, pero flat_map lo hace en un paso.

```
# Cada palabra la separamos en letras  
palabras = ["hola", "mundo"]  
palabras.map { |p| p.chars }  
# => [["h", "o", "l", "a"], ["m", "u", "n", "d", "o"]]  
  
palabras.flat_map { |p| p.chars }  
# => ["h", "o", "l", "a", "m", "u", "n", "d", "o"]
```

ANALOGÍA: map es como pedirle a cada invitado que traiga su plato en una charola: terminas con una charola por invitado. flat_map es como decir "traigan sus platos y pónganlos todos en la mesa grande": un solo nivel, todo junto.

Ejemplo real: tienes usuarios, cada usuario tiene varios pedidos, quieres la lista de TODOS los pedidos:

```

usuarios = [
  { nombre: "Ana", pedidos: [101, 102] },
  { nombre: "Luis", pedidos: [103] },
  { nombre: "María", pedidos: [104, 105, 106] }
]

todos_los_pedidos = usuarios.flat_map { |u| u[:pedidos] }
# => [101, 102, 103, 104, 105, 106]

```

Con `map` habrías obtenido `[[101, 102], [103], [104, 105, 106]]` y luego necesitarías `flatten`. `flat_map` es exactamente `map + flatten(1)` en un solo paso, y además es más eficiente porque no crea el arreglo intermedio.

ERROR COMÚN: usar `flat_map` cuando el bloque no devuelve arreglos. Si el bloque devuelve valores simples, `flat_map` se comporta igual que `map`. No hace daño, pero confunde a quien lee tu código: `flat_map` promete "voy a aplanar algo", y si no hay nada que aplanar, mejor usa `map`.

Encadenar enumerables: pensar en tuberías

Aquí está la verdadera magia. Cada método enumerable devuelve una colección, así que puedes encadenarlos: la salida de uno es la entrada del siguiente. Los datos fluyen como por una tubería, transformándose en cada etapa.

```

ventas = [
  { producto: "pan", monto: 25, mes: "enero" },
  { producto: "leche", monto: 30, mes: "enero" },
  { producto: "queso", monto: 120, mes: "febrero" },
  { producto: "pan", monto: 25, mes: "febrero" },
  { producto: "vino", monto: 200, mes: "febrero" }
]

# ¿Cuánto se vendió en febrero, solo productos de más de $50?
total = ventas
  .select { |v| v[:mes] == "febrero" }
  .select { |v| v[:monto] > 50 }
  .sum { |v| v[:monto] }

puts total # => 320 (queso 120 + vino 200)

```

Lee la cadena como una receta: "de las ventas, quédate con las de febrero, de esas quédate con las de más de 50, y suma sus montos". Cada línea es un paso claro. Compáralo con la versión imperativa:

MALO (versión imperativa, difícil de leer):

```
total = 0
ventas.each do |v|
  if v[:mes] == "febrero"
    if v[:monto] > 50
      total = total + v[:monto]
    end
  end
end
```

BUENO (versión en tubería, se lee como español):

```
total = ventas
  .select { |v| v[:mes] == "febrero" }
  .select { |v| v[:monto] > 50 }
  .sum { |v| v[:monto] }
```

La versión en tubería dice QUÉ quieres, no CÓMO lo calculas paso a paso. Esa es la esencia de la programación funcional que Ruby abraza: describes el resultado, no los pasos mecánicos.

Otro ejemplo, un reporte completo:

```
# Top 3 productos más vendidos (por monto total)
top = ventas
  .group_by { |v| v[:producto] }
  .transform_values { |lista| lista.sum { |v| v[:monto] } }
  .sort_by { |producto, total| -total }
  .first(3)
  .to_h

p top # => {"vino"=>200, "queso"=>120, "leche"=>30}
```

Agrupar por producto, sumar los montos de cada grupo, ordenar descendente, tomar los 3 primeros, convertir a hash. Cinco pasos, cada uno clarísimo. Esto en un lenguaje sin enumerables serían 30 líneas de ciclos anidados.

BUENA PRÁCTICA: cuando una cadena se vuelve muy larga (más de 4 o 5 eslabones), considera partirla en variables con nombres descriptivos. Una cadena larguísima en una sola expresión es difícil de depurar: si algo sale mal, no sabes en qué eslabón. Partirla te permite inspeccionar cada paso con p.

```
febrero = ventas.select { |v| v[:mes] == "febrero" }  
caras = febrero.select { |v| v[:monto] > 50 }  
total = caras.sum { |v| v[:monto] }
```

Mismo resultado, pero ahora puedes imprimir febrero y caras para verificar cada etapa. En código real, la legibilidad le gana a la elegancia compacta.

lazy: cuando los datos son infinitos o gigantes

Todos los enumerables que hemos visto son "ansiosos" (eager): procesan TODA la colección de inmediato. Eso está bien con 100 elementos, pero ¿qué pasa con un millón? ¿O con una secuencia infinita?

```
# Esto intentaría crear un arreglo infinito y colgaría tu programa:  
# (1..Float::INFINITY).map { |n| n * 2 }.first(5) # ¡NO LO HAGAS!
```

lazy convierte la tubería en "perezosa": no calcula nada hasta que se lo pides, y solo calcula lo necesario.

```
pares_dobles = (1..Float::INFINITY).lazy.map { |n| n * 2 }  
puts pares_dobles.first(5).to_a.inspect # => [2, 4, 6, 8, 10]
```

Con .lazy, el map no se ejecuta de inmediato: se queda esperando. Cuando pides .first(5), solo calcula los primeros 5 elementos y se detiene. Infinito domado.

ANALOGÍA: los enumerables normales son como un restaurante que cocina TODO el menú antes de que llegues, aunque solo vayas a pedir una ensalada. lazy es como un restaurante normal: cocina solo lo que pides, cuando lo pides.

Uso real con archivos gigantes: procesar un archivo de 10 GB línea por línea sin cargarlo todo en memoria:

```
# Sin lazy: lee TODO el archivo a memoria (puede tronar)
# lineas = File.readlines("gigante.log")

# Con lazy: procesa de a una línea
File.foreach("gigante.log").lazy
  .select { |linea| linea.include?("ERROR") }
  .first(10)
  .each { |linea| puts linea }
```

File.foreach ya lee línea por línea, y .lazy se asegura de que el select no intente procesar todo antes de entregar. Con esto encuentras los primeros 10 errores de un archivo gigantesco usando casi nada de memoria.

No necesitas lazy todos los días, pero cuando lo necesites, es la diferencia entre un programa que funciona y uno que se queda sin memoria.

PROYECTO: agenda de contactos en consola

Llegó el momento de juntar todo. Vas a construir una agenda de contactos completa que corre en la terminal, con menú interactivo, las cuatro operaciones básicas y persistencia en archivo. Este es tu proyecto final del libro: tómallo en serio y al final tendrás un programa real que mostrar.

Primero, el diseño. Antes de escribir código, piensa en las piezas:

1. Un Contacto: nombre, teléfono, email. Lo representamos como un hash o como una clase. Usaremos una clase porque ya sabes POO y queda más limpio.
2. Una Agenda: maneja la lista de contactos y las operaciones (agregar, listar, buscar, eliminar). También se encarga de guardar y cargar desde archivo.
3. Un menú en consola: el usuario elige opciones, el programa responde. Ciclo hasta que elija salir.
4. Persistencia: guardar los contactos en un archivo (usaremos un formato simple, un archivo de texto con Marshal o mejor: un archivo con un contacto por línea en formato separado por pipes, para que sea legible).

Decisión de diseño: ¿cómo guardamos los datos? Opciones: Marshal (binario, rápido pero ilegible), JSON (legible, ya lo viste en el libro), o texto plano con separadores (simple y legible). Para este proyecto usaremos texto plano con | como separador: cada línea es "nombre|teléfono|email". Es simple, legible con cualquier editor, y te obliga a practicar split y join, dos métodos que debes dominar.

Estructura del programa (un solo archivo agenda.rb para simplicidad):

- [1] Clase Contacto: datos y validaciones básicas
- [2] Clase Agenda: lista en memoria + operaciones CRUD
- [3] Persistencia: guardar y cargar desde agenda.txt
- [4] Menú interactivo: loop con opciones
- [5] Arranque: cargar datos, mostrar menú, guardar al salir

Vamos por partes. Primero la clase Contacto:

```
class Contacto
  attr_accessor :nombre, :telefono, :email

  def initialize(nombre, telefono, email)
    @nombre = nombre.strip
    @telefono = telefono.strip
    @email = email.strip
  end

  def to_s
    "#{@nombre} | #{@telefono} | #{@email}"
  end

  def coincide?(texto)
    texto = texto.downcase
    @nombre.downcase.include?(texto) ||
    @telefono.include?(texto) ||
    @email.downcase.include?(texto)
  end
end
```

Fíjate en los detalles: strip quita espacios accidentales al inicio y fin (el usuario suele dejar espacios). to_s define cómo se muestra el contacto. coincide? busca el texto en cualquiera de los tres campos, sin importar mayúsculas: es la base de la función de buscar.

Ahora la clase Agenda con las operaciones:

```
class Agenda
  ARCHIVO = "agenda.txt"

  def initialize
    @contactos = []
    cargar
  end
```

```

def agregar(contacto)
  @contactos << contacto
  guardar
end

def listar
  @contactos
end

def buscar(texto)
  @contactos.select { |c| c.coincide?(texto) }
end

def eliminar(indice)
  return nil unless indice.between?(0, @contactos.length - 1)
  eliminado = @contactos.delete_at(indice)
  guardar
  eliminado
end

def total
  @contactos.length
end

private

def guardar
  File.open(ARCHIVO, "w") do |f|
    @contactos.each do |c|
      f.puts "#{c.nombre}|#{c.telefono}|#{c.email}"
    end
  end
end

def cargar
  return unless File.exist?(ARCHIVO)
  File.foreach(ARCHIVO) do |linea|
    nombre, telefono, email = linea.chomp.split("|", 3)
    @contactos << Contacto.new(nombre, telefono, email)
  end
end
end

```

Decisiones explicadas: ARCHIVO es una constante porque el nombre del archivo no cambia. agregar guarda inmediatamente después de agregar: así nunca pierdes datos aunque se vaya la luz. eliminar también guarda. between? valida el índice de forma legible. En cargar, split("|", 3) limita a 3 partes por si el email contuviera un pipe (raro, pero seguro). Los métodos guardar y cargar son private porque son detalles internos: el usuario de la clase solo necesita agregar, listar, buscar y eliminar.

Nota el patrón: la Agenda carga al iniciar y guarda en cada cambio. Es simple y robusto para un programa de consola. En una app más grande usarías una base de datos, pero para este proyecto el archivo es perfecto y te enseña el ciclo completo de persistencia.

Ahora el menú interactivo:

```
def mostrar_menu
  puts "\n=== AGENDA DE CONTACTOS ==="
  puts "1. Agregar contacto"
  puts "2. Listar contactos"
  puts "3. Buscar contacto"
  puts "4. Eliminar contacto"
  puts "5. Salir"
  print "Elige una opción: "
  gets.chomp
end

def pedir_dato(mensaje)
  print "#{mensaje}: "
  gets.chomp
end
```

Y el programa principal que une todo:

```

agenda = Agenda.new

loop do
  case mostrar_menu
  when "1"
    nombre = pedir_dato("Nombre")
    telefono = pedir_dato("Teléfono")
    email = pedir_dato("Email")
    if nombre.empty?
      puts "El nombre no puede estar vacío."
    else
      agenda.agregar(Contacto.new(nombre, telefono, email))
      puts "Contacto agregado."
    end
  when "2"
    contactos = agenda.listar
    if contactos.empty?
      puts "No hay contactos todavía."
    else
      contactos.each_with_index do |c, i|
        puts "#{i + 1}. #{c}"
      end
    end
  when "3"
    texto = pedir_dato("Texto a buscar")
    resultados = agenda.buscar(texto)
    if resultados.empty?
      puts "Sin resultados."
    else
      resultados.each { |c| puts c }
    end
  when "4"
    contactos = agenda.listar
    contactos.each_with_index { |c, i| puts "#{i + 1}. #{c}" }
    num = pedir_dato("Número a eliminar").to_i
    eliminado = agenda.eliminar(num - 1)
    puts eliminado ? "Eliminado: #{eliminado.nombre}" : "Número inválido."
  when "5"
    puts "¡Hasta luego!"
    break
  else
    puts "Opción inválida."
  end
end
end

```

Detalles a notar: el loop con `case` es el esqueleto clásico de un menú de consola. `gets.chomp` lee la opción. En eliminar, restamos 1 porque el usuario ve números desde 1 pero el arreglo empieza en 0. Validamos el nombre vacío al agregar (un contacto sin nombre no sirve). El `break` sale del loop. Como la Agenda guarda en cada cambio, al salir no hay nada más que hacer: los datos ya están a salvo.

En la sección de ejemplos vas a ver el programa COMPLETO en un solo bloque, listo para copiar, guardar como `agenda.rb` y correr con `ruby agenda.rb`. También verás una versión mejorada con colores usando la gema `colorize` de la unidad anterior: las dos unidades trabajando juntas, como debe ser.

Qué es Rails y por qué Ruby es famoso por él

Ruby existía desde 1995, pero el mundo lo descubrió en 2004 cuando un programador danés llamado David Heinemeier Hansson (DHH) liberó Ruby on Rails, el framework web que había creado para la empresa Basecamp. Rails demostró que se podían construir aplicaciones web potentes con una fracción del código y del tiempo que tomaba en Java o PHP. Twitter, GitHub, Shopify, Airbnb y miles de empresas más empezaron con Rails. Por eso, cuando alguien dice "quiero aprender Ruby", el 90% de las veces su meta real es "quiero aprender Rails para conseguir trabajo o lanzar mi proyecto".

Un framework es como un esqueleto prefabricado para construir aplicaciones. Sin framework, para hacer una app web tendrías que programar desde cero cómo recibir peticiones HTTP, cómo conectarte a la base de datos, cómo organizar tus archivos. Rails te da todo eso resuelto con convenciones sensatas: te dice dónde va cada cosa y te da herramientas para las tareas comunes. Tú te concentras en lo único de tu aplicación.

La filosofía de Rails se resume en dos frases: "Convención sobre configuración" y "No te repitas" (DRY, Don't Repeat Yourself). Convención sobre configuración significa que Rails asume decisiones razonables por ti: si tienes un modelo llamado Producto, Rails asume que su tabla en la base de datos se llama productos, que su controlador se llama ProductosController, que sus vistas están en `app/views/productos`. No tienes que configurar nada de eso; solo sigue las convenciones y todo se conecta solo. DRY significa que cada pieza de conocimiento vive en un solo lugar: defines algo una vez y Rails lo reutiliza en todas partes.

ANALOGÍA: programar web sin framework es como construir una casa empezando por fabricar tus propios ladrillos, tus clavos y tu cemento. Rails es como llegar a un terreno donde ya están los cimientos, las paredes levantadas y la instalación eléctrica: tú decides los acabados, los muebles y el color de las paredes, que es lo que hace única a tu casa. El 80% del trabajo repetitivo ya está hecho.

El patrón MVC explicado con un restaurante

Todas las aplicaciones Rails (y la mayoría de las apps web modernas) se organizan con el patrón MVC: Modelo, Vista, Controlador. Es una forma de dividir responsabilidades para que el código no sea un espagueti. La mejor forma de entenderlo es con la analogía del restaurante:

- El MODELO es la cocina. Ahí están los ingredientes (los datos) y las recetas (las reglas del negocio). La cocina sabe cómo preparar cada platillo, qué ingredientes lleva, y se encarga de guardar todo en la alacena (la base de datos). El cliente nunca entra a la cocina, pero todo lo que come sale de ahí. En Rails, los modelos son clases Ruby que representan tus datos (Usuario, Producto, Pedido) y saben cómo guardarlos, validarlos y relacionarlos.
- La VISTA es el platillo servido en la mesa: lo que el cliente ve. El plato bonito, la presentación, el menú impreso. La vista no cocina nada ni decide nada: solo muestra. En Rails, las vistas son plantillas (normalmente HTML con código Ruby incrustado) que muestran los datos al usuario en el navegador.
- El CONTROLADOR es el mesero. Recibe tu pedido (la petición del navegador), va a la cocina y dice "un Producto número 5", la cocina lo prepara (el modelo busca en la base de datos), el mesero lo recoge y lo lleva a tu mesa presentándolo bonito (elige qué vista mostrar con esos datos). El mesero no cocina ni decora platos: coordina. En Rails, los controladores reciben las peticiones web, hablan con los modelos y deciden qué vista mostrar.

El flujo completo: escribes una URL en tu navegador (haces un pedido) -> el controlador lo recibe (el mesero toma la orden) -> le pide datos al modelo (va a la cocina) -> el modelo consulta la base de datos (busca en la alacena) -> el controlador recibe los datos y elige una vista (el mesero recoge el platillo) -> la vista presenta los datos como página web (el platillo llega a tu mesa).

¿Por qué dividir así? Porque cada pieza se puede cambiar sin romper las demás. ¿Quieres cambiar cómo se ve la página? Tocas solo la vista. ¿Cambió una regla del negocio? Tocas solo el modelo. ¿Necesitas una nueva página? Agregas un método al controlador y una vista. Sin MVC, todo estaría mezclado y cualquier cambio sería una aventura.

Cuando generes tu primer scaffold en Rails al final de esta unidad, vas a ver estas tres piezas creadas automáticamente: el modelo, las vistas y el controlador, cada uno en su carpeta. Y vas a entender exactamente qué hace cada uno gracias al restaurante.

Instalar Rails y crear tu primera aplicación

Instalar Rails es instalar una gema, como aprendiste en la unidad 9. Pero es una gema grande con muchas dependencias, así que toma unos minutos.

```
gem install rails
```

Verifica la instalación:

```
rails --version
```

Deberías ver algo como Rails 8.x.x (la versión exacta depende de cuándo leas esto; no te preocupes, los conceptos son los mismos).

NOTA IMPORTANTE: Rails funciona mejor en macOS y Linux. En Windows, la forma recomendada es usar WSL (Windows Subsystem for Linux), que te da un Linux dentro de Windows. Si estás en Windows sin WSL, la instalación puede darte dolores de cabeza; con WSL, funciona igual que en Linux. Este es un consejo práctico que te ahorra horas: no pelees con Rails en Windows nativo.

Ahora crea tu primera aplicación. En la terminal, ve a la carpeta donde guardas tus proyectos y ejecuta:

```
rails new mi_tienda
```

Rails va a crear una carpeta `mi_tienda` con toda la estructura y va a instalar las gemas necesarias con Bundler (¿ves cómo todo lo de la unidad 9 se conecta?). Esto toma unos minutos la primera vez. Al terminar, entra a la carpeta:

```
cd mi_tienda
```

Vamos a ver la estructura de carpetas, porque entenderla es entender Rails:

- `app/`: aquí vive TU código. Lo más importante:
- `app/models/`: los modelos (la cocina).
- `app/views/`: las vistas (los platillos presentados).

- `app/controllers/`: los controladores (los meseros).
- `app/assets/`: imágenes, estilos CSS, JavaScript.
- `config/`: configuración. Aquí está `config/routes.rb`, el archivo que define qué URLs existen y a qué controlador van (como el menú del restaurante: lista qué platillos se pueden pedir).
- `db/`: la base de datos. Aquí viven las migraciones, que son archivos Ruby que describen cambios en la estructura de la base de datos (crear tablas, agregar columnas).
- `Gemfile`: las gemas del proyecto (¡hola de nuevo, unidad 9!).
- `public/`: archivos estáticos.

No necesitas memorizar todo hoy. Lo importante es el mapa mental: tu código va en `app/`, dividido en `models`, `views` y `controllers`. Todo lo demás es infraestructura que Rails maneja por ti.

Levantar el servidor y ver la página de bienvenida

Este es el momento mágico. Dentro de la carpeta de tu aplicación, ejecuta:

```
bin/rails server
```

O la forma corta que todo mundo usa:

```
rails s
```

Vas a ver algo como:

```
=> Booting Puma
=> Rails 8.x.x application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Listening on http://127.0.0.1:3000
```

Rails levantó un servidor web en tu propia computadora, en el puerto 3000. Ahora abre tu navegador y visita:

`http://localhost:3000`

Vas a ver la página de bienvenida de Rails, con su logo y la frase de que tu aplicación está corriendo. Ese momento, ver tu primera app Rails viva en el navegador, es el que convierte a muchos en desarrolladores web. Tómate un segundo para disfrutarlo: hay un servidor web corriendo en tu máquina, escrito en Ruby, y tú lo levantaste con un comando.

Para detener el servidor, presiona Ctrl+C en la terminal.

¿Qué está pasando detrás? Cuando visitas localhost:3000, tu navegador hace una petición HTTP a tu propia computadora. El servidor Puma (incluido con Rails) la recibe y se la pasa a Rails. Rails mira config/routes.rb, ve que la ruta raíz muestra la página de bienvenida, y te la devuelve. Todavía no hay modelos ni base de datos involucrados: es solo Rails saludándote. Pero la tubería completa (navegador -> servidor -> Rails -> respuesta) ya funciona.

ERROR COMÚN: "no puedo ver la página". Verifica tres cosas: que el servidor siga corriendo en la terminal (si cerraste la terminal o presionaste Ctrl+C, se detuvo), que la URL sea exactamente http://localhost:3000 (el puerto 3000 es clave), y que no haya otro programa usando el puerto 3000 (si ves error de "address already in use", otro servidor lo está ocupando; ciérralo o usa rails s -p 3001 para otro puerto).

Tu primer scaffold: un CRUD en segundos

Recuerda el CRUD de tu agenda de contactos: crear, leer, actualizar, eliminar. En Rails, generar esas cuatro operaciones para un recurso es UN comando: el scaffold. Es la demostración más impresionante de la filosofía de Rails.

Asegúrate de estar en la carpeta mi_tienda y ejecuta:

```
rails generate scaffold Producto nombre:string precio:decimal
```

Rails genera: el modelo Producto, una migración para crear la tabla productos, el controlador ProductosController con las acciones CRUD, las vistas (listado, formulario, detalle), y las rutas. Todo siguiendo las convenciones. Después corre la migración para crear la tabla en la base de datos:

```
rails db:migrate
```

Reinicia el servidor (rails s) y visita http://localhost:3000/productos. Vas a ver una página funcional donde puedes crear productos, verlos, editarlos y eliminarlos. Un CRUD completo, generado en segundos, con código limpio y organizado en MVC.

Párate a pensar en lo que acabas de hacer: con dos comandos creaste lo que a mano te habría tomado días (y lo que en tu agenda de consola te tomó una unidad completa del libro). Esa es la razón por la que Rails existe: las partes repetitivas de las aplicaciones web ya están resueltas; tú te concentras en lo que hace especial a tu idea.

Mira lo que generó, porque cada pieza refleja el MVC del restaurante:

- `app/models/producto.rb`: el modelo. La cocina: sabe que un producto tiene nombre y precio, y cómo guardarse en la base de datos.
- `app/controllers/productos_controller.rb`: el controlador. El mesero: tiene acciones `index` (listar), `show` (ver uno), `new/create` (crear), `edit/update` (editar), `destroy` (eliminar).
- `app/views/productos/`: las vistas. Los platillos: `index.html.erb` (la lista), `show.html.erb` (el detalle), `_form.html.erb` (el formulario reutilizable).
- `db/migrate/..._create_productos.rb`: la migración que creó la tabla.
- `config/routes.rb`: ahora tiene `resources :productos`, que declara todas las URLs del CRUD de un plumazo.

No tienes que entender cada línea hoy. Pero abre esos archivos y échales un ojo: vas a reconocer Ruby por todas partes (es tu lenguaje ahora), mezclado con convenciones de Rails. Cada archivo que no entiendas es una puerta a tu siguiente aprendizaje.

Qué sigue después de este libro

Terminaste el libro, pero esto es el inicio, no el final. Aquí va tu mapa de ruta:

1. Profundiza en Rails. Este libro te dio el lenguaje; ahora aprende el framework en serio. El tutorial oficial (guides.rubyonrails.org) es excelente y gratuito. Construye una app real: un blog, una tienda, un sistema de tareas. La única forma de aprender Rails es construyendo.
2. Aprende Git y GitHub. Si aún no usas control de versiones, es tu siguiente paso obligatorio. Todo trabajo profesional usa Git. Guarda tus proyectos ahí: es tu portafolio.
3. Bases de datos en serio. Rails usa SQLite por defecto (perfecta para aprender), pero en producción verás PostgreSQL. Aprende SQL más a fondo: el libro te dio la base, ahora practica consultas reales.
4. JavaScript para el frontend. Rails maneja el backend de maravilla, pero las interfaces modernas interactivas necesitan JavaScript. Con Ruby + Rails + JavaScript cubres todo el desarrollo web.

5. Construye y publica. Sube tus proyectos a internet (Render, Fly.io y Heroku tienen planes gratuitos o baratos para Rails). Un proyecto publicado vale más que diez tutoriales terminados. Tu academia, lucasegaacademy.com, es el ejemplo perfecto de a dónde puedes llegar.
6. La comunidad Ruby es tu aliada. RubyGems para gemas, foros, meetups, código abierto en GitHub. Cuando te atores (y te vas a atorar, todos nos atoramos), pregunta: la comunidad Ruby tiene fama de ser de las más amables.

Mira hacia atrás: empezaste sin saber qué era una variable y ahora puedes construir una agenda funcional, escribir pruebas, usar expresiones regulares, instalar gemas con Bundler y levantar una aplicación Rails. Eso no es poco: es la base completa de un desarrollador Ruby. Lo que sigue es práctica, práctica y más práctica. El libro se acaba; tu carrera empieza.

Migraciones: la base de datos como código

Cuando generaste el scaffold, Rails creó también un archivo de migración en `db/migrate/`. Ábrelo y verás algo así:

```
class CreateProductos < ActiveRecord::Migration[8.0]
  def change
    create_table :productos do |t|
      t.string :nombre
      t.decimal :precio

      t.timestamps
    end
  end
end
```

Esto es Ruby que describe la estructura de la base de datos: crea la tabla `productos` con una columna `nombre` de texto, una columna `precio` decimal, y `t.timestamps` agrega automáticamente `created_at` y `updated_at` (fecha de creación y última modificación de cada registro).

La idea revolucionaria: la estructura de tu base de datos vive en código, versionada junto con tu proyecto. Si mañana necesitas agregar una columna `stock` a `productos`, no alteras la base de datos a mano: generas una migración:

```
rails generate migration AgregarStockAProductos stock:integer
```

Rails crea el archivo, corres rails db:migrate, y la columna existe. Si algo sale mal, rails db:rollback deshace la última migración. Y si un compañero clona tu proyecto, con rails db:migrate obtiene la base de datos idéntica a la tuya, sin que le pases ningún archivo SQL.

ANALOGÍA: antes de las migraciones, cambiar la base de datos era como remodelar una casa sin planos: cada quien hacía cambios y nadie sabía cómo quedó. Las migraciones son los planos versionados: cada cambio queda registrado en orden, cualquiera puede replicar la casa exacta, y puedes volver atrás si una remodelación sale mal.

Tipos de columna comunes: string (texto corto), text (texto largo), integer, decimal (para dinero, evita errores de redondeo de los flotantes), boolean, datetime, references (para relacionar tablas, como "este pedido pertenece a un usuario").

ERROR COMÚN: editar una migración que YA se aplicó (que ya corriste con db:migrate) y esperar que la base de datos cambie. No funciona así: Rails marca las migraciones aplicadas y no las vuelve a correr. Si necesitas cambiar algo, crea una NUEVA migración. Editar migraciones viejas solo causa que tu base de datos y la de tus compañeros queden diferentes sin que nadie entienda por qué.

Rutas: el menú del restaurante

El archivo config/routes.rb es el menú del restaurante: lista qué URLs existen y a qué controlador van. Cuando generaste el scaffold, Rails agregó una línea:

```
Rails.application.routes.draw do
  resources :productos
end
```

Esa sola línea, resources :productos, crea SIETE rutas automáticamente:

- GET /productos -> index (listar todos, el mesero trae el menú completo)
- GET /productos/new -> new (formulario para uno nuevo)
- POST /productos -> create (guardar el nuevo)
- GET /productos/:id -> show (ver uno específico, el platillo número 5)
- GET /productos/:id/edit -> edit (formulario para editar)
- PATCH /productos/:id -> update (guardar los cambios)
- DELETE /productos/:id -> destroy (eliminar)

Estas siete acciones son la convención REST, el estándar de la web para CRUD. No tienes que memorizarlas hoy, pero entiende la idea: con una palabra (resources) Rails cableó todo el CRUD web. Si visitas /productos ves la lista; si visitas /productos/3 ves el producto 3; el :id es un parámetro que cambia según lo que pidas.

Puedes ver todas las rutas de tu app con:

```
rails routes
```

Te muestra una tabla con cada URL, el verbo HTTP y la acción del controlador que la atiende. Cuando en el futuro agregues cosas como una página de inicio o un buscador, las declararás aquí. Por ejemplo, para que la página principal sea tu lista de productos:

```
root "productos#index"
```

Eso dice: "cuando visiten /, atiéndelos con la acción index del controlador de productos". Ahora <http://localhost:3000> muestra directamente tus productos, sin tener que escribir /productos.

La belleza del sistema: las URLs tienen sentido, los controladores tienen acciones predecibles, y todo sigue la misma convención en todas las apps Rails del mundo. Cuando entres a trabajar a una empresa con una app Rails, ya sabrás dónde buscar cada cosa antes de ver el código.

El ecosistema: qué más trae Rails bajo el capó

Ya viste modelos, vistas, controladores, migraciones y rutas. Pero Rails trae más herramientas que irás descubriendo, y vale la pena que sepas que existen:

- Active Record: la capa que conecta tus modelos con la base de datos. Te permite escribir `Producto.where("precio > ?", 1000)` en Ruby en vez de SQL a mano. Es elegante y potente, aunque aprender SQL (como hiciste en este libro) te hace mejor usándolo, porque entiendes qué está pasando debajo.
- Action View con ERB: las vistas usan archivos `.html.erb` donde mezclas HTML con Ruby entre `<% %>`. Por ejemplo: `<% @productos.each do |p| %> ... <% end %>`. Ya conoces la sintaxis; solo cambia el contexto.
- Los helpers y los partials: formas de reutilizar pedazos de vista (como el formulario `_form.html.erb` que viste, que se usa tanto para crear como para editar: DRY en acción).

- La consola de Rails: con rails console abres un irb conectado a tu aplicación, con acceso a tus modelos y a la base de datos real. Puedes hacer `Producto.create(nombre: "Laptop", precio: 15000)` y verlo aparecer en tu app. Es la herramienta de experimentación y depuración que más vas a usar.
- Los generadores: rails generate no solo hace scaffolds. Hay generadores para modelos sueltos, controladores sueltos, migraciones, mailers. Cada uno crea los archivos en su lugar correcto siguiendo las convenciones. Úsalos siempre en vez de crear archivos a mano: te ahorran errores de ubicación y nombres.
- El modo desarrollo vs producción: cuando corres rails estás en desarrollo, con recarga automática de código y errores detallados en el navegador. En producción todo se optimiza y los errores se ocultan al usuario. Esa distinción existe en todos los frameworks serios.

No necesitas aprender todo esto esta semana. Pero ahora sabes el mapa del territorio: cuando escuches "Active Record" o "migración" o "ERB", ya no son palabras mágicas, son piezas que encajan en el rompecabezas que acabas de armar.

Jugando con rails console: tu app como laboratorio

Recuerda irb, tu laboratorio de la unidad 9. Rails tiene su propia versión conectada a tu aplicación: la consola de Rails. Con el servidor detenido (o en otra terminal), ejecuta dentro de tu proyecto:

```
rails console
```

O corto: rails c. Te abre un irb donde tus modelos existen y están conectados a la base de datos real. Prueba esto con tu app biblioteca:

```
Producto.count
# => 0 (si aún no creaste ninguno desde la web)

p1 = Producto.create(nombre: "Laptop", precio: 15000)
# => #<Producto id: 1, nombre: "Laptop", precio: 15000, ...>

p2 = Producto.create(nombre: "Mouse", precio: 300)

Producto.count
# => 2

Producto.where("precio > ?", 1000)
# => [#<Producto id: 1, nombre: "Laptop", ...>]

p1.update(precio: 14000)
p1.precio
# => 14000

p2.destroy
Producto.count
# => 1
```

Acabas de hacer CRUD completo desde la consola, sin tocar el navegador. Esto es oro para depurar: ¿la página no muestra lo que esperas? Abre la consola y verifica qué hay realmente en la base de datos. ¿Un método del modelo se comporta raro? Pruébalo aquí con datos reales antes de culpar a la vista.

Fíjate en `Producto.where("precio > ?", 1000)`: es Ruby que genera SQL por ti. El `?` es un placeholder seguro: Rails pone el 1000 ahí escapándolo correctamente, lo que te protege de inyección SQL (un ataque donde datos maliciosos se cuelan en tus consultas). Nunca construyas consultas pegando strings con datos del usuario; usa siempre los placeholders. Es un hábito de seguridad que te va a acompañar toda tu carrera.

La consola también es donde los profesionales prueban ideas de negocio antes de escribir vistas: "¿cómo obtengo los productos más caros de cada categoría?" lo prototipas aquí con Active Record, y cuando la consulta funciona, la pasas al controlador. El mismo flujo que aprendiste con irb en tus programas de consola, ahora aplicado a aplicaciones web.

Para salir: `exit`, igual que en irb.

Lo que Rails NO hace por ti (expectativas realistas)

El scaffold impresiona tanto que es fácil pensar que Rails lo hace todo. Mejor poner los pies en la tierra desde hoy:

1. Rails no diseña tu aplicación. Te da el CRUD genérico, pero las reglas de TU negocio (¿quién puede ver qué? ¿qué pasa cuando se agota el inventario? ¿cómo se calcula el envío?) las programas tú en los modelos y controladores.
2. Rails no hace que tu app se vea bonita. Las vistas generadas son funcionales pero feas: HTML sin estilo. Aprender CSS (o un framework como Bootstrap o Tailwind) sigue siendo tu responsabilidad si quieres que se vea profesional.
3. Rails no te libra de entender la web. Conceptos como HTTP, peticiones GET vs POST, sesiones y cookies siguen ahí debajo. Rails los maneja por ti, pero cuando algo falla a ese nivel, necesitas saber qué está pasando.
4. Rails no escala tu lógica de negocio. Para apps pequeñas y medianas es rapidísimo de desarrollar. Para sistemas gigantescos necesitarás aprender sobre caché, trabajos en segundo plano y optimización de base de datos. Pero eso es un problema del futuro: primero construye algo que la gente use.

La forma correcta de ver a Rails: es un multiplicador de tu productividad, no un reemplazo de tu criterio. Entre más Ruby sepas (y ahora sabes bastante), más provecho le sacas. El scaffold te da el 80% mecánico en segundos; el 20% que hace única a tu aplicación, ese lo pones tú. Y ese 20% es exactamente lo que los clientes y empleadores pagan.

Ejemplos paso a paso

Ejemplo 1: reduce para estadísticas de ventas. Este programa calcula total, promedio y el día de mayor venta de una semana, todo con reduce y tuberías. Es el tipo de reporte que te piden en un trabajo real.


```

ventas_semana = [1200, 850, 1500, 900, 2100, 1800, 950]
dias = %w[lunes martes miércoles jueves viernes sábado domingo]

total = ventas_semana.reduce(0, :+)
promedio = total.to_f / ventas_semana.length
mejor_dia = dias.zip(ventas_semana).reduce do |mejor, actual|
  actual[1] > mejor[1] ? actual : mejor
end

puts "Total: ${total}"
puts "Promedio diario: ${promedio.round(2)}"
puts "Mejor día: #{mejor_dia[0]} con ${mejor_dia[1]}"

```

Explicación: `reduce(0, :+)` suma todo. `zip` combina días con ventas en pares [día, monto]. El segundo `reduce` compara pares y se queda con el de mayor monto: el acumulador siempre guarda "el mejor hasta ahora". Es el patrón de "encontrar el máximo con criterio" que aparece en todas partes. Nota cómo `reduce` sirve para mucho más que sumar: aquí encuentra un elemento ganador.

Ejemplo 2: `group_by` para un reporte de calificaciones. Agrupa estudiantes por rango de calificación y cuenta cuántos hay en cada grupo. Típico reporte escolar.

```

estudiantes = [
  { nombre: "Ana", calif: 9 },
  { nombre: "Luis", calif: 5 },
  { nombre: "María", calif: 8 },
  { nombre: "Pedro", calif: 4 },
  { nombre: "Sofía", calif: 10 },
  { nombre: "Diego", calif: 7 }
]

def rango(calif)
  case calif
  when 9..10 then "Excelente (9-10)"
  when 7..8 then "Bien (7-8)"
  when 6 then "Suficiente (6)"
  else "Reprobado (<6)"
  end
end

reporte = estudiantes
  .group_by { |e| rango(e[:calif]) }
  .transform_values { |grupo| grupo.map { |e| e[:nombre] } }

reporte.each do |rango_nombre, nombres|
  puts "#{rango_nombre}: #{nombres.join(", ")} (#{nombres.length})"
end

```

Explicación: primero `group_by` organiza por rango usando la función auxiliar. Luego `transform_values` convierte cada grupo (arreglo de hashes) en solo la lista de nombres. El resultado es un reporte legible. Esta combinación (`group_by` + `transform_values`) es una de las más usadas en análisis de datos con Ruby: apréndetela de memoria.

Ejemplo 3: `sort_by` con múltiples criterios. Ordena una lista de películas por calificación descendente y, en empates, por año ascendente (la más vieja primero).

```

películas = [
  { titulo: "El Padrino", calif: 9.2, año: 1972 },
  { titulo: "Batman", calif: 9.0, año: 2008 },
  { titulo: "Matrix", calif: 8.7, año: 1999 },
  { titulo: "Interestelar", calif: 8.7, año: 2014 },
  { titulo: "Coco", calif: 8.4, año: 2017 }
]

ordenadas = películas.sort_by { |p| [-p[:calif], p[:año]] }

ordenadas.each do |p|
  puts "#{p[:titulo]} (#{p[:año]}) - #{p[:calif]}"
end

```

Explicación: la llave de ordenamiento es un arreglo [-calif, año]. El signo menos invierte el orden de la calificación (mayor primero). En empates de calificación (Matrix e Interestelar, ambas 8.7), se compara el segundo elemento: año ascendente, así que Matrix (1999) va antes que Interestelar (2014). Ordenar por múltiples criterios con un arreglo es un truco elegante que te ahorra comparadores complicados.

Ejemplo 4: flat_map para aplanar pedidos. Una tienda tiene clientes, cada cliente tiene pedidos, cada pedido tiene productos. Queremos la lista plana de todos los productos vendidos.

```

clientes = [
  { nombre: "Ana", pedidos: [["pan", "leche"], ["queso"]] },
  { nombre: "Luis", pedidos: [["vino"]] },
  { nombre: "María", pedidos: [["pan"], ["leche", "vino", "queso"]] }
]

productos = clientes.flat_map { |c| c[:pedidos] }.flatten
puts "Productos vendidos: #{productos.join(", ")}"
puts "Total de productos: #{productos.length}"

# ¿Cuáles productos se vendieron más? (combinando con group_by y sort_by)
top = productos.group_by { |p| p }
  .transform_values(&:length)
  .sort_by { |prod, cant| -cant }

puts "\nRanking:"
top.each { |prod, cant| puts "  #{prod}: #{cant}" }

```

Explicación: `flat_map` aplanar un nivel (junta los pedidos de todos los clientes), y `flatten` aplanar el segundo nivel (los productos dentro de cada pedido). Después, `group_by` cuenta ocurrencias y `sort_by` las ordena: un mini análisis de ventas en pocas líneas. Observa el `&:length`: es un atajo para `{ |x| x.length }`. Cuando el bloque solo llama un método sin argumentos, puedes usar `&::nombre_metodo`. Espera, sin escapar: `&:length`. Este atajo es muy idiomático en Ruby.

Ejemplo 5: lazy con una secuencia infinita. Genera los primeros 10 números primos usando una secuencia infinita perezosa. Sin lazy, esto sería imposible.

```
def primo?(n)
  return false if n < 2
  (2..Math.sqrt(n)).none? { |d| n % d == 0 }
end

primos = (2..Float::INFINITY).lazy.select { |n| primo?(n) }.first(10)
puts "Primeros 10 primos: #{primos.to_a.join(", ")}"
```

Explicación: `(2..Float::INFINITY)` es un rango infinito. `.lazy` hace que `select` no intente evaluar todo el infinito (lo cual colgaría el programa), sino que vaya probando número por número hasta juntar los 10 primeros primos. `.first(10)` detiene la búsqueda en cuanto los tiene. Es la demostración perfecta de la evaluación perezosa: solo se calcula lo necesario, ni un número más.

Ejemplo 6: tubería completa de análisis de texto. Dado un párrafo, encuentra las 5 palabras más frecuentes (ignorando palabras cortas). Combina casi todo lo de la unidad.

```
texto = "Ruby es un lenguaje elegante y Ruby es divertido. " \
        "Programar en Ruby se siente natural y elegante. " \
        "La comunidad Ruby es amable y la sintaxis es elegante."

top_palabras = texto
  .downcase
  .scan(/[a-záéíóúñü]+)/
  .reject { |p| p.length < 4 }
  .group_by { |p| p }
  .transform_values(&:length)
  .sort_by { |palabra, veces| -veces }
  .first(5)

puts "Top 5 palabras:"
top_palabras.each { |palabra, veces| puts "  #{palabra}: #{veces} veces" }
```

Explicación: la tubería hace: minúsculas, extraer palabras con regex (incluyendo acentos y ñ), quitar palabras de menos de 4 letras, agrupar por palabra, contar cada grupo, ordenar por frecuencia descendente, tomar las 5 primeras. Siete eslabones, cada uno obvio por separado, y juntos resuelven un análisis de texto real. Este ejemplo usa regex de la unidad 9 y enumerables de la unidad 10: todo el libro trabajando junto.

Ejemplo 7: PROYECTO COMPLETO - agenda de contactos funcional. Copia este código en un archivo llamado agenda.rb y ejecútalo con ruby agenda.rb. Es el proyecto final: menú interactivo, CRUD completo y persistencia en archivo.

```
# agenda.rb - Agenda de contactos en consola
# Proyecto final del Curso Completo de Ruby - Lucas EGA Academy

class Contacto
  attr_accessor :nombre, :telefono, :email

  def initialize(nombre, telefono, email)
    @nombre = nombre.strip
    @telefono = telefono.strip
    @email = email.strip
  end

  def to_s
    "#{@nombre} | #{@telefono} | #{@email}"
  end

  def coincide?(texto)
    texto = texto.downcase
    @nombre.downcase.include?(texto) ||
    @telefono.include?(texto) ||
    @email.downcase.include?(texto)
  end
end

class Agenda
  ARCHIVO = "agenda.txt"

  def initialize
    @contactos = []
    cargar
  end

  def agregar(contacto)
    @contactos << contacto
    guardar
  end
end
```

```

end

def listar
  @contactos
end

def buscar(texto)
  @contactos.select { |c| c.coincide?(texto) }
end

def eliminar(indice)
  return nil unless indice.between?(0, @contactos.length - 1)
  eliminado = @contactos.delete_at(indice)
  guardar
  eliminado
end

def total
  @contactos.length
end

private

def guardar
  File.open(ARCHIVO, "w") do |f|
    @contactos.each do |c|
      f.puts "#{c.nombre}|#{c.telefono}|#{c.email}"
    end
  end
end

def cargar
  return unless File.exist?(ARCHIVO)
  File.foreach(ARCHIVO) do |linea|
    nombre, telefono, email = linea.chomp.split("|", 3)
    @contactos << Contacto.new(nombre, telefono, email)
  end
end

def mostrar_menu
  puts "\n=== AGENDA DE CONTACTOS ==="
  puts "1. Agregar contacto"
  puts "2. Listar contactos"
  puts "3. Buscar contacto"
end

```

```

    puts "4. Eliminar contacto"
    puts "5. Salir"
    print "Elige una opción: "
    gets.chomp
end

def pedir_dato(mensaje)
  print "#{mensaje}: "
  gets.chomp
end

agenda = Agenda.new
puts "Bienvenido a tu agenda. Tienes #{agenda.total} contactos guardados."

loop do
  case mostrar_menu
  when "1"
    nombre = pedir_dato("Nombre")
    telefono = pedir_dato("Teléfono")
    email = pedir_dato("Email")
    if nombre.empty?
      puts "El nombre no puede estar vacío."
    else
      agenda.agregar(Contacto.new(nombre, telefono, email))
      puts "Contacto agregado correctamente."
    end
  when "2"
    contactos = agenda.listar
    if contactos.empty?
      puts "No hay contactos todavía. Agrega el primero con la opción 1."
    else
      puts "\n--- Tus contactos (#{contactos.length}) ---"
      contactos.each_with_index { |c, i| puts "#{i + 1}. #{c}" }
    end
  when "3"
    texto = pedir_dato("Texto a buscar")
    resultados = agenda.buscar(texto)
    if resultados.empty?
      puts "Sin resultados para '#{texto}'."
    else
      puts "\n--- Resultados (#{resultados.length}) ---"
      resultados.each { |c| puts c }
    end
  when "4"
    contactos = agenda.listar

```

```

    if contactos.empty?
      puts "No hay contactos para eliminar."
    else
      contactos.each_with_index { |c, i| puts "#{i + 1}. #{c}" }
      num = pedir_dato("Número a eliminar").to_i
      eliminado = agenda.eliminar(num - 1)
      if eliminado
        puts "Eliminado: #{eliminado.nombre}"
      else
        puts "Número inválido."
      end
    end
  end
when "5"
  puts "¡Hasta luego! Tus contactos están guardados."
  break
else
  puts "Opción inválida. Elige un número del 1 al 5."
end
end
end

```

Explicación: este es el programa completo y funcional. La clase Contacto modela cada persona con sus tres datos y sabe buscar dentro de sí misma (coincide?). La clase Agenda es el cerebro: mantiene el arreglo en memoria, implementa las cuatro operaciones CRUD, y persiste en agenda.txt automáticamente en cada cambio (nunca pierdes datos). El menú es un loop con case que traduce la elección del usuario en acciones. Pruébalo: agrega tres contactos, cierra el programa, vuelve a abrirlo y ahí siguen. Eso es persistencia real. Como ejercicio de graduación, agrega la opción 6 "Editar contacto" por tu cuenta: ya tienes todas las piezas.

Ejemplo 8: agenda mejorada con colores (integrando la unidad 9). Si instalaste colorize, esta versión del menú se ve profesional. Solo cambian las funciones de presentación; la lógica es la misma.


```

require "colorize"

def mostrar_menu
  puts "\n" + "=== AGENDA DE CONTACTOS ===".bold.blue
  puts "1. Agregar contacto"
  puts "2. Listar contactos"
  puts "3. Buscar contacto"
  puts "4. Eliminar contacto"
  puts "5. Salir"
  print "Elige una opción: "
  gets.chomp
end

def exito(mensaje)
  puts mensaje.green
end

def error(mensaje)
  puts mensaje.red.bold
end

def aviso(mensaje)
  puts mensaje.yellow
end

# Uso dentro del programa principal:
# exito("Contacto agregado correctamente.")
# error("El nombre no puede estar vacío.")
# aviso("Sin resultados para '#{texto}'.")

```

Explicación: separamos la presentación en funciones pequeñas (exito, error, aviso) en vez de regar `.green` y `.red` por todo el código. Si mañana quieres cambiar cómo se ven los errores, lo cambias en un solo lugar. Este es el principio DRY aplicado a la presentación. Además, nota cómo las unidades del libro se combinan: la gema `colorize` de la unidad 9 mejora el proyecto de la unidad 10. Así trabajan los programadores reales: cada cosa que aprenden se suma a su caja de herramientas y la usan en el siguiente proyecto.

Práctica

1. Usa `reduce` para calcular el producto de todos los números del 1 al 10 (debe dar 3628800). Luego calcula la suma de los cuadrados de `[1, 2, 3, 4, 5]` (debe dar 55).

SOLUCIÓN EXPLICADA: para el producto, el valor inicial debe ser 1 (no 0, porque todo por cero es cero) y la operación es *. Para la suma de cuadrados, el bloque eleva cada número al cuadrado antes de acumular.

```
producto = (1..10).reduce(1, :*)
puts producto # => 3628800

suma_cuadrados = [1, 2, 3, 4, 5].reduce(0) { |acc, n| acc + n**2 }
puts suma_cuadrados # => 55 (1 + 4 + 9 + 16 + 25)
```

El detalle del valor inicial es clave: para multiplicar empiezas en 1 (el neutro de la multiplicación), para sumar en 0 (el neutro de la suma). Si empiezas el producto en 0, el resultado siempre será 0. Este error es clásico y la forma de evitarlo es preguntarte: "¿qué valor no cambia el resultado al combinarlo?".

2. Tienes una lista de empleados con su departamento. Agrúpalos por departamento con `group_by` y luego muestra cuántos hay en cada uno.

```
empleados = [
  { nombre: "Ana", depto: "ventas" },
  { nombre: "Luis", depto: "sistemas" },
  { nombre: "María", depto: "ventas" },
  { nombre: "Pedro", depto: "rh" },
  { nombre: "Sofía", depto: "sistemas" },
  { nombre: "Diego", depto: "ventas" }
]
```

SOLUCIÓN EXPLICADA: `group_by` con el departamento como criterio, y luego contamos cada grupo con `length`. `transform_values` convierte cada lista en su conteo.

```
por_depto = empleados.group_by { |e| e[:depto] }
conteo = por_depto.transform_values(&:length)
conteo.each { |depto, n| puts "#{depto}: #{n} empleados" }
# ventas: 3 empleados
# sistemas: 2 empleados
# rh: 1 empleado
```

Fíjate en `&:length` como atajo de `{ |grupo| grupo.length }`. Este patrón (agrupar y contar) aparece en reportes de todo tipo: ventas por mes, errores por tipo, usuarios por país. Domínalo.

3. Ordena esta lista de libros por año de publicación descendente (los más nuevos primero). En caso de empate en el año, ordena por título alfabéticamente.

```
libros = [
  { titulo: "Cien años de soledad", año: 1967 },
  { titulo: "Rayuela", año: 1963 },
  { titulo: "La casa de los espíritus", año: 1982 },
  { titulo: "Crónica de una muerte anunciada", año: 1981 },
  { titulo: "El amor en los tiempos del cólera", año: 1985 },
  { titulo: "El otoño del patriarca", año: 1975 }
]
```

SOLUCIÓN EXPLICADA: la llave es `[-año, titulo]`. El `menos` invierte el año (descendente) y el título como segundo elemento desempata alfabéticamente (ascendente, que es el orden natural de los strings).

```
ordenados = libros.sort_by { |l| [-l[:año], l[:titulo]] }
ordenados.each { |l| puts "#{l[:año]} - #{l[:titulo]}" }
# 1985 - El amor en los tiempos del cólera
# 1982 - La casa de los espíritus
# 1981 - Crónica de una muerte anunciada
# 1975 - El otoño del patriarca
# 1967 - Cien años de soledad
# 1963 - Rayuela
```

El truco del arreglo como llave de ordenamiento es una de las técnicas más elegantes de Ruby: cada posición del arreglo es un criterio, en orden de prioridad.

4. Tienes una lista de frases. Usa `flat_map` para obtener todas las palabras de todas las frases en un solo arreglo plano, y luego encuentra la palabra más larga con `reduce`.

```
frases = ["Ruby es elegante", "me encanta programar", "los enumerables son poderosos"]
```

SOLUCIÓN EXPLICADA: `flat_map` con `split` divide cada frase en palabras y aplana todo en un solo arreglo. Luego `reduce` compara longitudes quedándose con la más larga.

```

palabras = frases.flat_map { |f| f.split }
p palabras # => ["Ruby", "es", "elegante", "me", "encanta", "programar", "los",
"enumerables", "son", "poderosos"]

mas_larga = palabras.reduce { |acc, p| p.length > acc.length ? p : acc }
puts mas_larga # => "enumerables"

```

Con map habrías obtenido un arreglo de arreglos y necesitado flatten después. flat_map lo hace en un paso. Y el reduce para el máximo es el patrón "quedarse con el mejor según un criterio" que ya viste en los ejemplos.

5. Escribe una tubería que, dada una lista de productos con precio y categoría, calcule el precio promedio de los productos de la categoría "electrónica" que cuesten más de \$1000.

```

productos = [
  { nombre: "Laptop", precio: 15000, categoria: "electrónica" },
  { nombre: "Mouse", precio: 300, categoria: "electrónica" },
  { nombre: "Silla", precio: 2000, categoria: "muebles" },
  { nombre: "Teclado", precio: 1200, categoria: "electrónica" },
  { nombre: "Mesa", precio: 3500, categoria: "muebles" },
  { nombre: "Monitor", precio: 4500, categoria: "electrónica" }
]

```

SOLUCIÓN EXPLICADA: filtramos por categoría, filtramos por precio, y promediamos. Para el promedio necesitamos la suma y el conteo; sum con bloque nos da la suma.

```

seleccionados = productos
  .select { |p| p[:categoria] == "electrónica" }
  .select { |p| p[:precio] > 1000 }

if seleccionados.empty?
  puts "No hay productos que cumplan los criterios."
else
  total = seleccionados.sum { |p| p[:precio] }
  promedio = total.to_f / seleccionados.length
  puts "Promedio: ${promedio.round(2)}"
  # => Promedio: $6900.0 (15000 + 1200 + 4500 = 20700 / 3)
end

```

Nota la validación de empty?: si ningún producto cumple, dividir entre cero daría error. Siempre que calcules un promedio sobre un filtrado, considera el caso de que el filtrado quede vacío. Es un caso borde clásico.

6. Agrega a la agenda del Ejemplo 7 una opción 6 "Editar contacto": muestra la lista numerada, pide el número, luego pide los nuevos datos (si el usuario deja un campo vacío, se conserva el valor anterior). Implementa el método actualizar en la clase Agenda.

SOLUCIÓN EXPLICADA: el método actualizar reemplaza el contacto en el índice dado y guarda. En el menú, para cada campo pedimos el nuevo valor y usamos empty? para decidir si conservar el anterior.

```
# Dentro de la clase Agenda, agregar:
def actualizar(indice, contacto)
  return nil unless indice.between?(0, @contactos.length - 1)
  @contactos[indice] = contacto
  guardar
  contacto
end
```

```
# En el menú, opción "6":
when "6"
  contactos = agenda.listar
  if contactos.empty?
    puts "No hay contactos para editar."
  else
    contactos.each_with_index { |c, i| puts "#{i + 1}. #{c}" }
    num = pedir_dato("Número a editar").to_i - 1
    original = agenda.listar[num]
    if original.nil?
      puts "Número inválido."
    else
      nombre = pedir_dato("Nuevo nombre (vacío = conservar '#{original.nombre}')" )
      telefono = pedir_dato("Nuevo teléfono (vacío = conservar)")
      email = pedir_dato("Nuevo email (vacío = conservar)")
      nuevo = Contacto.new(
        nombre.empty? ? original.nombre : nombre,
        telefono.empty? ? original.telefono : telefono,
        email.empty? ? original.email : email
      )
      agenda.actualizar(num, nuevo)
      puts "Contacto actualizado."
    end
  end
end
```

El operador ternario (condición ? valor_si : valor_no) es perfecto aquí: compacto y legible para "si está vacío usa el anterior, si no usa el nuevo". Con esto tu agenda tiene el CRUD verdaderamente completo: crear, leer, actualizar y eliminar.

7. Crea una pequeña aplicación Rails siguiendo los pasos de la unidad: instala rails, crea una app llamada biblioteca, genera un scaffold Libro con titulo:string y autor:string, corre la migración, levanta el servidor y visita /libros. Describe qué ves en cada paso.

SOLUCIÓN EXPLICADA: estos son los comandos, en orden, cada uno en la terminal:

```
gem install rails
rails --version
rails new biblioteca
cd biblioteca
rails generate scaffold Libro titulo:string autor:string
rails db:migrate
rails s
```

Luego en el navegador visitas <http://localhost:3000/libros>. Lo que debes ver: una página titulada "Libros" con un enlace "New libro" (o "Nuevo libro" según la versión). Al dar clic, un formulario con campos para título y autor. Al guardar, el libro aparece en la lista, y cada libro tiene enlaces para verlo (Show), editarlo (Edit) y eliminarlo (Destroy). Todo eso lo generó Rails: el modelo Libro, el controlador LibrosController con sus 7 acciones, las vistas y las rutas. Si algo falla, revisa: ¿estás dentro de la carpeta biblioteca? ¿Corriste db:migrate antes de levantar el servidor? ¿El servidor sigue corriendo? Esos tres son el 90% de los problemas.

8. Explica con tus propias palabras, usando la analogía del restaurante, qué pasa cuando un usuario visita <http://localhost:3000/libros/5> en tu app biblioteca. Menciona el rol del controlador, el modelo y la vista.

SOLUCIÓN EXPLICADA: no hay una única respuesta correcta, pero debe contener estas piezas. Ejemplo de respuesta válida:

"El usuario hace un pedido: quiere ver el libro número 5 (como pedir 'el platillo número 5 del menú'). La petición llega al mesero, que es LibrosController en su acción show. El mesero va a la cocina y pide el libro con id 5: el modelo Libro lo busca en la base de datos (la alacena). Si lo encuentra, el mesero lo lleva a la mesa: elige la vista show.html.erb, que es el platillo presentado, y muestra el título y el autor del libro en una página bonita. Si el libro no existe, la cocina dice 'no hay' y el mesero devuelve un error 404, como decir 'ese platillo no está en el menú'. El controlador nunca cocina (no toca la base de datos directamente), el modelo nunca decora platos (no genera HTML), y la vista nunca toma pedidos (no procesa URLs). Cada quien hace su parte."

Si puedes explicar esto sin mirar tus apuntes, entendiste MVC. Y si entiendes MVC, entiendes la arquitectura de casi todas las aplicaciones web modernas, no solo Rails.

9. Desafío final integrador: escribe un programa que combine todo lo del libro. El programa debe: (1) leer un archivo de texto `ventas.txt` donde cada línea tiene formato `producto|precio|cantidad`, (2) usar una clase `Venta` para representar cada línea, (3) calcular con una tubería de enumerables el ingreso total por producto (`precio * cantidad` sumado por producto), (4) ordenar los productos por ingreso descendente con `sort_by`, (5) mostrar el reporte con `colorize` (el top 1 en verde y negritas), y (6) incluir al menos 3 pruebas minitest para la clase `Venta`. Usa este archivo de ejemplo:

```
Laptop|15000|2
Mouse|300|10
Laptop|15000|1
Teclado|1200|5
Mouse|300|4
```

SOLUCIÓN EXPLICADA: este ejercicio es tu examen final del libro: archivos, clases, enumerables encadenados, `sort_by`, gemas y pruebas, todo junto. La clase `Venta` parsea cada línea y calcula su subtotal. La tubería agrupa por producto, suma subtotales, ordena y muestra.

```

# ventas.rb
require "colorize"

class Venta
  attr_reader :producto, :precio, :cantidad

  def initialize(linea)
    @producto, precio_txt, cantidad_txt = linea.chomp.split("|")
    @precio = precio_txt.to_f
    @cantidad = cantidad_txt.to_i
  end

  def subtotal
    @precio * @cantidad
  end
end

ventas = File.readlines("ventas.txt").map { |linea| Venta.new(linea) }

reporte = ventas
  .group_by(&:producto)
  .transform_values { |lista| lista.sum(&:subtotal) }
  .sort_by { |producto, total| -total }

puts "=== REPORTE DE VENTAS ===".bold.blue
reporte.each_with_index do |(producto, total), i|
  linea = "#{i + 1}. #{producto}: ${total}"
  puts i == 0 ? linea.green.bold : linea
end

```



```
# test_venta.rb
require "minitest/autorun"
require_relative "ventas_sin_main"

class TestVenta < Minitest::Test
  def test_parsea_linea
    v = Venta.new("Laptop|15000|2")
    assert_equal "Laptop", v.producto
    assert_equal 15000, v.precio
    assert_equal 2, v.cantidad
  end

  def test_subtotal
    v = Venta.new("Mouse|300|10")
    assert_equal 3000, v.subtotal
  end

  def test_convierte_tipos
    v = Venta.new("Teclado|1200|5")
    assert_kind_of Float, v.precio
    assert_kind_of Integer, v.cantidad
  end
end
```

Nota el detalle: para que las pruebas puedan cargar la clase sin ejecutar el reporte, el código del reporte debería estar protegido con `if __FILE__ == $0` (solo se ejecuta cuando corres el archivo directamente, no cuando lo cargas con `require_relative`). Es un truco profesional que separa "definir" de "ejecutar":

```
if __FILE__ == $0
  ventas = File.readlines("ventas.txt").map { |linea| Venta.new(linea) }
  # ... resto del reporte ...
end
```

Con eso, `test_venta.rb` carga solo la clase y el reporte solo corre cuando ejecutas `ventas.rb` directamente. Si resolviste este ejercicio sin atorarte mucho, estás listo: sabes leer archivos, modelar con clases, transformar con enumerables, presentar con gemas y verificar con pruebas. Eso es programar en Ruby de verdad.

Resumen de la unidad

- reduce (alias inject) condensa una colección en un solo valor usando un acumulador. La última línea del bloque debe devolver el nuevo acumulador. Para sumas simples usa `.reduce(0, :+)` o `.sum`.
- `group_by` organiza elementos en un hash según el criterio del bloque. Combinado con `transform_values` es la base de reportes y análisis de datos.
- `sort_by` ordena por una llave calculada. Para múltiples criterios usa un arreglo como llave: `sort_by { |e| [-e[:puntos], e[:nombre]] }`.
- `flat_map` transforma y aplanar en un paso: ideal cuando cada elemento produce una lista y quieres un solo arreglo plano.
- Encadena enumerables como tuberías: `select -> map -> sort_by -> sum`. Describe qué quieres, no los pasos mecánicos. Si la cadena es muy larga, pártela en variables con nombre para poder depurar.
- `lazy` hace perezosa una tubería: solo calcula lo necesario. Úsalo con secuencias infinitas o archivos gigantescos que no caben en memoria.
- Un proyecto real se diseña antes de codificarse: identifica las clases, las operaciones y la persistencia. La agenda usa Contacto (datos), Agenda (CRUD + archivo) y un menú con loop/case.
- Guarda los datos en cada cambio, no solo al salir: así nunca pierdes información aunque el programa se cierre mal.
- Rails es el framework web de Ruby: convención sobre configuración y DRY. Te da la estructura para que te concentres en lo único de tu app.
- MVC con el restaurante: el Modelo es la cocina (datos y reglas), la Vista es el platillo servido (lo que ve el usuario), el Controlador es el mesero (coordina pedidos entre ambos).
- Flujo Rails: `rails new app`, `rails generate scaffold Recurso campos`, `rails db:migrate`, `rails s`, y visitas `http://localhost:3000`. El scaffold genera un CRUD completo en segundos.
- Después del libro: profundiza en Rails con los tutoriales oficiales, aprende Git, practica SQL, suma JavaScript para el frontend, publica tus proyectos y apóyate en la comunidad Ruby.
- Felicidades: empezaste sin saber qué era una variable y terminaste construyendo una agenda funcional con persistencia, escribiendo pruebas, dominando regex y levantando tu primera app Rails. Eres programador Ruby. Lo que sigue es práctica.

EXAMEN FINAL

Instrucciones: este examen cubre todo el contenido del libro "Curso Completo de Ruby". Las preguntas 1 a 20 son de opción múltiple (elige a, b, c o d). Las preguntas 21 a 35 son de verdadero o falso. Las preguntas 36 a 50 son abiertas: escribe código, completa programas, predice salidas o corrige errores. Para aprobar necesitas al menos 35 respuestas correctas. ¡Suerte!

Un consejo antes de empezar: haz el examen con honestidad, sin copiar las respuestas ni correr el código para las preguntas de "predice la salida". El examen no es para mí ni para nadie más: es para ti. Si haces trampa, el único engañado eres tú, porque saldrás creyendo que dominas algo que no dominas, y eso se nota en cuanto escribes código real. En cambio, si lo haces a conciencia, cada error se convierte en un mapa exacto de lo que debes repasar. Tómate tu tiempo, lee cada pregunta dos veces y contesta primero las que te sepas seguro.

Guía de repaso rápido antes del examen

Antes de empezar, repasa estos puntos clave de cada unidad. Si alguno te suena borroso, vuelve a su unidad antes de contestar.

Unidad 1 - Fundamentos: ``puts`` imprime con salto de línea, ``print`` sin salto. Las variables se crean con ``=`` y no necesitan declaración de tipo. Los tipos básicos son Integer, Float, String, Symbol, true/false y nil. La interpolación ``#{variable}`` solo funciona con comillas dobles. La división entre enteros trunca: ``7 / 2`` es 3. ``irb`` es tu consola interactiva para probar código al instante.

Unidad 2 - Control de flujo: ``if/elsif/else/end`` para decisiones, ``unless`` como ``if`` negado, ``case/when`` para múltiples opciones. Bucles: ``while`` y ``until`` repiten mientras se cumpla (o no) una condición; ``times``, ``each`` y los rangos ``(1..10)`` son la forma idiomática de repetir un número conocido de veces. Cuidado con los bucles infinitos: siempre asegúrate de que la condición pueda volverse falsa.

Unidad 3 - Arreglos y hashes: los índices empiezan en 0. ``push`` y ``<<`` agregan al final, ``pop`` saca del final, ``unshift`/`shift`` trabajan al principio. ``each`` itera sin modificar, ``map`` devuelve un arreglo nuevo transformado, ``select`` filtra, ``length`/`size`` cuentan elementos. Los hashes guardan pares clave-valor; las claves suelen ser símbolos (``{ nombre: "Ana" }``) y se acceden con ``hash[:clave]``. Un hash devuelve ``nil`` si pides una clave inexistente.

Unidad 4 - Métodos y bloques: ``def nombre(param = default)`` define métodos; el valor de retorno es la última expresión evaluada (retorno implícito). Los bloques son fragmentos de código entre ``{}`` o ``do...end`` que se pasan a métodos como ``each``, ``map`` o ``times``. ``yield`` ejecuta dentro del método el bloque que recibió. Los procs y lambdas son bloques guardados en variables.

Unidad 5 - POO: ``class`` define clases, ``initialize`` es el constructor que se ejecuta con ``new``, las variables de instancia (``@nombre``) viven dentro del objeto, y ``attr_reader``/``attr_writer``/``attr_accessor`` generan los métodos de acceso. Todo en Ruby es un objeto, hasta los números.

Unidad 6 - Herencia y módulos: ``class Hija < Padre`` declara herencia; ``super`` llama al método del padre. Los módulos agrupan métodos reutilizables y se mezclan con ``include`` (métodos de instancia) o ``extend`` (métodos de clase). Ruby no tiene herencia múltiple de clases, pero los mixins cubren esa necesidad.

Unidad 7 - Excepciones: ``begin/rescue/ensure`` atrapa errores; ``rescue`` puede especificar el tipo de error (``rescue ArgumentError``); ``ensure`` siempre se ejecuta; ``raise`` lanza tus propias excepciones. Nunca dejes un ``rescue`` vacío que oculte errores.

Unidad 8 - Archivos: ``File.open("f.txt", "w")`` escribe, ``"r"`` lee, ``"a"`` agrega al final. Con bloque se cierra solo. ``each_line`` recorre línea por línea, ``read`` lee todo de golpe. Los errores de archivo inexistente son ``Errno::ENOENT``.

Unidad 9 - Regex y strings: las expresiones regulares van entre ``/ /``; ``=~`` y ``match?`` prueban coincidencias; ``^`` es inicio, ``$`` es fin, ``.`` es cualquier carácter. ``gsub`` reemplaza todas las ocurrencias, ``sub`` solo la primera, ``split`` divide, ``include?`` busca substrings, ``downcase``/``upcase`` cambian mayúsculas.

Unidad 10 - Ecosistema: las gemas se instalan con ``gem install``, Bundler gestiona dependencias con el Gemfile, ``ri`` y la documentación en ruby-doc.org resuelven dudas, y las pruebas con Minitest o RSpec son el estándar profesional.

Mapa de temas por pregunta

Para que ubiques rápido qué repasar si fallas: preguntas 1-4 (fundamentos y tipos), 5-7 (condicionales y bucles), 8-11 (arreglos y hashes), 12-14 (métodos y bloques), 15-18 (POO, herencia y módulos), 19-20 (excepciones y archivos), 21-23 (truthiness, símbolos, objetos), 24-27 (each, alcance, retorno implícito, puts/print), 28-31 (módulos, interpolación, super, gemas), 32-35 (regex, encapsulamiento, ensure, times), 36-40 (métodos, while,

each_with_index, promedio, argumentos), 41-45 (clases, retorno, hashes, validación, herencia con super), 46-50 (gsub/split, módulos, archivos con rescue, select/each, FizzBuzz).

PREGUNTA 1: ¿Cuál es la forma correcta de imprimir "Hola Ruby" en la pantalla?

a) print_line("Hola Ruby") b) puts "Hola Ruby" c) echo "Hola Ruby" d) console.log("Hola Ruby")

PREGUNTA 2: ¿Qué valor tiene la variable `x` después de ejecutar `x = 7 / 2` en Ruby?

a) 3.5 b) 3 c) 4 d) Error de sintaxis

PREGUNTA 3: ¿Cuál de las siguientes es la forma correcta de crear un símbolo en Ruby?

a) "nombre" b) 'nombre' c) :nombre d) #nombre

PREGUNTA 4: ¿Qué imprime el siguiente código? `puts 5 > 3 && 2 < 4`

a) false b) true c) nil d) Error

PREGUNTA 5: ¿Cuál es la salida de este código?

```
x = 10
if x > 5
  puts "mayor"
else
  puts "menor"
end
```

a) menor b) mayor c) 10 d) No imprime nada

PREGUNTA 6: ¿Qué hace el modificador `unless` en Ruby?

a) Ejecuta el bloque solo si la condición es verdadera b) Ejecuta el bloque solo si la condición es falsa c) Repite el bloque mientras la condición sea verdadera d) Es un sinónimo de `while`

PREGUNTA 7: ¿Cuál es la salida de este código?

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

a) 1 2 3 (cada uno en una línea) b) 0 1 2 (cada uno en una línea) c) 3 3 3 (cada uno en una línea) d) Error: times no acepta bloques

PREGUNTA 8: Dado el arreglo `frutas = ["manzana", "pera", "uva"]`, ¿qué devuelve `frutas[1]`?

a) "manzana" b) "pera" c) "uva" d) nil

PREGUNTA 9: ¿Qué método se usa para agregar un elemento al FINAL de un arreglo?

a) unshift b) push (o el operador <<) c) insert d) append_first

PREGUNTA 10: Dado el hash `persona = { nombre: "Ana", edad: 25 }`, ¿qué devuelve `persona[:edad]`?

a) "edad" b) 25 c) "Ana" d) nil

PREGUNTA 11: ¿Qué hace el siguiente código?

```
numeros = [1, 2, 3, 4]
dobles = numeros.map { |n| n * 2 }
```

a) Modifica `numeros` para que valga [2, 4, 6, 8] b) Crea un nuevo arreglo [2, 4, 6, 8] sin modificar `numeros` c) Suma todos los elementos y devuelve 20 d) Devuelve [1, 2, 3, 4]

PREGUNTA 12: ¿Cuál es la forma correcta de definir un método que recibe un parámetro con valor por defecto?

a) def saluda(nombre = "mundo") b) def saluda(nombre := "mundo") c) def saluda(nombre default "mundo") d) def saluda(nombre ? "mundo")

PREGUNTA 13: En Ruby, ¿qué devuelve un método si no tiene una sentencia `return` explícita?

a) nil siempre b) El valor de la última expresión evaluada c) 0 d) Error: todo método necesita return

PREGUNTA 14: ¿Qué es un bloque en Ruby?

a) Un tipo de variable que guarda números b) Un fragmento de código entre do...end o llaves {} que se pasa a un método c) Un error que detiene el programa d) Un archivo de código externo

PREGUNTA 15: ¿Cuál es la salida de este código?

```
class Perro
  def initialize(nombre)
    @nombre = nombre
  end
  def ladra
    puts "#{@nombre} dice guau"
  end
end
fido = Perro.new("Fido")
fido.ladra
```

a) Error: @nombre no está definido b) "Fido dice guau" c) "nombre dice guau" d) nil

PREGUNTA 16: ¿Qué palabra clave se usa para que una clase herede de otra?

a) extends b) inherits c) < d) super

PREGUNTA 17: ¿Para qué sirve `attr\_reader :nombre` dentro de una clase?

a) Crea un método que permite leer (pero no escribir) el valor de @nombre b) Crea una variable global llamada nombre c) Impide que se lea el valor de @nombre d) Define un método que siempre devuelve nil

PREGUNTA 18: ¿Qué hace `include Enumerable` dentro de una clase?

a) Importa un archivo llamado enumerable.rb b) Mezcla (mixin) los métodos del módulo Enumerable en la clase c) Crea una herencia múltiple d) Nada, es un comentario especial

PREGUNTA 19: ¿Cuál es la estructura correcta para manejar una excepción en Ruby?

a) try / catch / finally b) begin / rescue / ensure c) attempt / recover / always d) do / rescue / end

PREGUNTA 20: ¿Qué hace el siguiente código?

```
File.open("datos.txt", "w") do |archivo|
  archivo.puts "Hola"
end
```

a) Lee el archivo datos.txt e imprime "Hola" b) Abre datos.txt en modo escritura, escribe "Hola" y cierra el archivo automáticamente c) Borra el archivo datos.txt d) Error: falta cerrar el archivo manualmente

PREGUNTA 21: Verdadero o falso: En Ruby, solo `false` y `nil` son valores falsos; todo lo demás (incluido el 0 y la cadena vacía "") se considera verdadero.

PREGUNTA 22: Verdadero o falso: Los símbolos (`:nombre`) y los strings ("`nombre`") son exactamente lo mismo y se pueden intercambiar en cualquier situación sin consecuencias.

PREGUNTA 23: Verdadero o falso: En Ruby todo es un objeto, incluso los números y los valores `true`, `false` y `nil` tienen métodos.

PREGUNTA 24: Verdadero o falso: El método `each` modifica el arreglo original y devuelve un arreglo nuevo con los cambios.

PREGUNTA 25: Verdadero o falso: Una variable local definida dentro de un método puede usarse fuera de ese método sin problema.

PREGUNTA 26: Verdadero o falso: El retorno implícito significa que un método devuelve automáticamente el valor de su última expresión evaluada, sin necesidad de escribir `return`.

PREGUNTA 27: Verdadero o falso: `puts` y `print` hacen exactamente lo mismo: imprimen texto sin agregar salto de línea al final.

PREGUNTA 28: Verdadero o falso: Un módulo en Ruby puede usarse para compartir métodos entre clases mediante `include`, sin necesidad de herencia.

PREGUNTA 29: Verdadero o falso: La interpolación de strings con `#{}` solo funciona en strings con comillas dobles, no en strings con comillas simples.

PREGUNTA 30: Verdadero o falso: `super` dentro de un método de una clase hija llama al método del mismo nombre definido en la clase padre.

PREGUNTA 31: Verdadero o falso: Una gema (gem) es un paquete de código Ruby distribuido y reutilizable que se instala con el comando `gem install`.

PREGUNTA 32: Verdadero o falso: En una expresión regular, el patrón `/^hola/` coincide con cualquier string que CONTenga la palabra "hola" en cualquier posición.

PREGUNTA 33: Verdadero o falso: Las variables de instancia (como `@nombre`) son visibles y accesibles directamente desde fuera del objeto sin necesidad de métodos.

PREGUNTA 34: Verdadero o falso: El bloque `ensure` dentro de un `begin/rescue` se ejecuta siempre, haya ocurrido una excepción o no.

PREGUNTA 35: Verdadero o falso: `5.times` ejecuta el bloque 5 veces pasando los valores 1, 2, 3, 4 y 5 a la variable del bloque.

PREGUNTA 36: Escribe un método llamado ``par_o_impar`` que reciba un número y devuelva el string "par" si el número es par o "impar" si es impar. (Pista: puedes usar el operador módulo %.)

PREGUNTA 37: Completa el siguiente programa para que imprima los números del 1 al 10, cada uno en su propia línea, usando un bucle ``while``:

```
contador = 1
# completa aquí
```

PREGUNTA 38: Predice la salida exacta del siguiente fragmento de código:

```
colores = ["rojo", "verde", "azul"]
colores.each_with_index do |color, i|
  puts "#{i}: #{color.upcase}"
end
```

PREGUNTA 39: Escribe un método llamado ``promedio`` que reciba un arreglo de números y devuelva su promedio (la suma dividida entre la cantidad de elementos). Si el arreglo está vacío, debe devolver 0.

PREGUNTA 40: El siguiente código tiene un error. Encuéntralo, explica por qué falla y escribe la versión corregida:

```
def saluda(nombre)
  puts "Hola " + nombre
end

saluda()
```

PREGUNTA 41: Crea una clase ``Rectangulo`` con un constructor que reciba ``base`` y ``altura``, y dos métodos: ``area`` (que devuelva base por altura) y ``perimetro`` (que devuelva 2 por base más 2 por altura). Luego crea un objeto de 5 por 3 e imprime su área.

PREGUNTA 42: Predice la salida exacta del siguiente código:

```
def misterio(n)
  return "cero" if n == 0
  n * 2
end

puts misterio(0)
puts misterio(5)
```

PREGUNTA 43: Escribe un programa que use un hash para contar cuántas veces aparece cada vocal en la palabra "programacion". El resultado debe ser un hash como `{ "a" => 2, "o" => 2, ... }` e imprimirlo.

PREGUNTA 44: Corrige el siguiente código para que maneje la posible excepción cuando el usuario ingresa algo que no es un número:

```
puts "Escribe tu edad:"
edad = gets.chomp.to_i
puts "En 10 años tendrás #{edad + 10}"
```

(Nota: `to\_i` convierte "abc" en 0 sin lanzar excepción, así que tu corrección debe validar la entrada de otra forma o usar `Integer().`.)

PREGUNTA 45: Escribe una clase `Vehiculo` con un método `describe` que imprima "Soy un vehículo", y una clase `Coche` que herede de `Vehiculo` y sobrescriba `describe` para imprimir "Soy un coche", llamando también al método del padre con `super` modificado para que el mensaje final sea "Soy un vehículo y soy un coche". (Pista: el padre debe devolver el string en lugar de imprimirlo, o usa `super` de forma creativa.)

PREGUNTA 46: Predice la salida exacta del siguiente código:

```
texto = "Ruby es genial"
puts texto.gsub(/[aeiou]/, "*")
puts texto.split(" ").length
```

PREGUNTA 47: Define un módulo llamado `Volador` con un método `volar` que imprima "Estoy volando", e inclúyelo en una clase `Pajaro` para que un objeto `Pajaro.new` pueda llamar a `volar`. Escribe el código completo.

PREGUNTA 48: Escribe un programa que lea el archivo "notas.txt" línea por línea e imprima solo las líneas que contengan la palabra "Ruby" (puede estar en mayúsculas o

minúsculas). Si el archivo no existe, debe imprimir "Archivo no encontrado" en lugar de romperse.

PREGUNTA 49: Predice la salida exacta del siguiente código:

```
suma = 0
[1, 2, 3, 4, 5].select { |n| n.odd? }.each { |n| suma += n }
puts suma
```

PREGUNTA 50: Escribe un método llamado `fizzbuzz` que reciba un número `n` e imprima los números del 1 al `n` con estas reglas: si el número es múltiplo de 3 imprime "Fizz", si es múltiplo de 5 imprime "Buzz", si es múltiplo de ambos imprime "FizzBuzz", y en cualquier otro caso imprime el número.

Respuestas y soluciones

RESPUESTA 1: b) `puts "Hola Ruby"` - `puts` es el método de Ruby para imprimir texto en pantalla con un salto de línea al final. La opción a) no existe en Ruby, c) `echo` es un comando de la terminal (no de Ruby), y d) `console.log` es de JavaScript.

RESPUESTA 2: b) 3 - En Ruby, la división entre dos enteros devuelve un entero (trunca el decimal, no redondea). `7 / 2` es 3. Si quisieras 3.5 tendrías que escribir `7 / 2.0` o `7.to_f / 2`. La opción a) sería correcta en lenguajes como Python 3 o JavaScript, pero no en Ruby.

RESPUESTA 3: c) `:nombre` - Los símbolos en Ruby empiezan con dos puntos. La opción a) y b) son strings (con comillas dobles y simples), y d) `#nombre` no es sintaxis válida para crear un símbolo (el `#` inicia comentarios).

RESPUESTA 4: b) true - `5 > 3` es verdadero y `2 < 4` es verdadero; el operador `&&` devuelve true solo si ambas condiciones son verdaderas. `puts` imprime el valor booleano como "true". Nota: Ruby también tiene los operadores `and` y `or` en palabras, pero `&&` y `||` son los preferidos porque tienen mayor precedencia y evitan sorpresas en expresiones combinadas.

RESPUESTA 5: b) mayor - Como `x` vale 10 y 10 es mayor que 5, se cumple la condición del `if` y se ejecuta `puts "mayor"`. La rama del `else` se ignora.

RESPUESTA 6: b) Ejecuta el bloque solo si la condición es falsa - ``unless`` es lo opuesto a ``if``: el código dentro se ejecuta cuando la condición NO se cumple. Es muy usado en Ruby para expresar condiciones negativas de forma legible, por ejemplo ``puts "error" unless valido``.

RESPUESTA 7: b) 0 1 2 (cada uno en una línea) - El método ``times`` ejecuta el bloque el número indicado de veces, pasando valores que empiezan en 0. ``3.times`` pasa 0, 1 y 2 a la variable ``i``. Es un error común esperar que empiece en 1.

RESPUESTA 8: b) "pera" - Los arreglos en Ruby usan índices que empiezan en 0: la posición 0 es "manzana", la 1 es "pera" y la 2 es "uva". La opción d) sería correcta si preguntáramos por ``frutas[5]``.

RESPUESTA 9: b) push (o el operador `<<`) - ``push`` agrega al final del arreglo y `<<` es su forma corta (``frutas << "mango"``). La opción a) ``unshift`` agrega al PRINCIPIO, c) ``insert`` agrega en una posición específica que debes indicar, y d) no existe. Detalle útil: `<<` devuelve el arreglo modificado, así que puedes encadenar: ``a = []; a << 1 << 2`` deja a ``a`` valiendo `[1, 2]`.

RESPUESTA 10: b) 25 - En el hash, la clave es el símbolo ``:edad`` y su valor es 25. La opción a) confunde la clave con el valor, c) es el valor de otra clave, y d) sería el resultado de pedir una clave inexistente como ``persona[:ciudad]``.

RESPUESTA 11: b) Crea un nuevo arreglo `[2, 4, 6, 8]` sin modificar ``numeros`` - El método ``map`` transforma cada elemento y devuelve un arreglo NUEVO; el original queda intacto. Si quisieras modificar el original usarías ``map!`` (con signo de admiración). La opción a) describe ``map!``, no ``map``. En la comunidad Ruby, los métodos terminados en ``!`` se llaman "bang methods" y por convención son las versiones peligrosas o mutantes de su contraparte sin ``!``: ``sort`` vs ``sort!``, ``gsub`` vs ``gsub!``, ``map`` vs ``map!``. Cuando veas el ``!``, pregúntate si de verdad quieres modificar el original.

RESPUESTA 12: a) ``def saluda(nombre = "mundo")`` - En Ruby los parámetros con valor por defecto se definen con el signo ``=``. Las opciones b), c) y d) no son sintaxis válida de Ruby.

RESPUESTA 13: b) El valor de la última expresión evaluada - Esto se llama "retorno implícito" y es una de las características más elegantes de Ruby. La opción a) solo sería cierta si el método estuviera vacío o su última expresión evaluara a nil. La d) es falsa: ``return`` es opcional.

RESPUESTA 14: b) Un fragmento de código entre `do...end` o llaves `{ }` que se pasa a un método - Los bloques son una de las características estrella de Ruby: permiten pasar

comportamiento a los métodos, como en `[1,2,3].each { |n| puts n }`. Las demás opciones describen cosas que no tienen nada que ver.

RESPUESTA 15: b) "Fido dice guau" - El método `initialize` es el constructor: cuando se crea `Perro.new("Fido")`, el string "Fido" se guarda en la variable de instancia `@nombre`. Luego `ladra` usa interpolación `"#{@nombre} dice guau"` para imprimir "Fido dice guau". La opción a) es incorrecta porque `@nombre` sí queda definido por el constructor.

RESPUESTA 16: c) `<` - En Ruby la herencia se declara con el signo menor que: `class Coche < Vehiculo`. La opción a) `extends` es de Java, b) no existe en Ruby, y d) `super` sirve para llamar al método del padre, no para declarar la herencia.

RESPUESTA 17: a) Crea un método que permite leer (pero no escribir) el valor de `@nombre` - `attr_reader` genera automáticamente el método lector `nombre` que devuelve `@nombre`. Si quisieras también escribir usarías `attr_writer`, y para ambas cosas `attr_accessor`. La opción c) describe lo contrario.

RESPUESTA 18: b) Mezcla (mixin) los métodos del módulo Enumerable en la clase - `include` mete los métodos del módulo dentro de la clase como métodos de instancia, sin herencia. Es la forma en que Ruby logra compartir comportamiento entre clases no relacionadas. La opción a) confunde `include` con `require`. Dato extra: si usaras `extend` en lugar de `include`, los métodos del módulo se agregarían como métodos de CLASE (para llamar como `MiClase.metodo`), no de instancia.

RESPUESTA 19: b) `begin / rescue / ensure` - Esta es la estructura de manejo de excepciones en Ruby: el código riesgoso va en `begin`, los errores se atrapan en `rescue` y el código que siempre debe ejecutarse va en `ensure`. La opción a) es de Java/JavaScript/C#, no de Ruby.

RESPUESTA 20: b) Abre datos.txt en modo escritura, escribe "Hola" y cierra el archivo automáticamente - La "w" significa modo escritura (crea el archivo o lo sobrescribe). Al pasar un bloque a `File.open`, Ruby cierra el archivo solo cuando el bloque termina, así no se te olvida cerrarlo. La opción d) es falsa precisamente por eso.

RESPUESTA 21: Verdadero - En Ruby solo `false` y `nil` son falsos. Esto sorprende a quienes vienen de otros lenguajes: el número 0 es verdadero, la cadena vacía "" es verdadera y el arreglo vacío [] también es verdadero. Por eso `if 0` ejecuta su bloque en Ruby, al contrario que en C o JavaScript.

RESPUESTA 22: Falso - Los símbolos y los strings NO son lo mismo. Un símbolo (`:nombre`) es un identificador inmutable y único en memoria: cada vez que escribes

``:nombre`` es el mismo objeto. Un string (``"nombre"``) crea un objeto nuevo cada vez (a menos que uses strings congelados). Por eso los símbolos se prefieren como claves de hash y nombres de métodos: son más rápidos y usan menos memoria. No siempre se pueden intercambiar sin consecuencias.

RESPUESTA 23: Verdadero - En Ruby absolutamente todo es un objeto: ``5.class`` devuelve Integer, ``"hola".class`` devuelve String, ``true.class`` devuelve TrueClass y ``nil.class`` devuelve NilClass. Por eso puedes llamar métodos sobre literales, como ``5.times`` o ``"hola".upcase``. No hay "tipos primitivos" como en Java.

RESPUESTA 24: Falso - ``each`` NO modifica el arreglo y NO devuelve un arreglo nuevo con cambios: simplemente ejecuta el bloque una vez por cada elemento y devuelve el arreglo ORIGINAL sin cambios. Si quieres transformar los elementos necesitas ``map`` (devuelve arreglo nuevo) o ``map!`` (modifica el original).

RESPUESTA 25: Falso - Las variables locales definidas dentro de un método solo existen dentro de ese método (su "alcance" o scope es el método). Fuera del método no existen y tratar de usarlas provoca un error de variable no definida. Para compartir datos entre el exterior y el método se usan parámetros y valores de retorno.

RESPUESTA 26: Verdadero - El retorno implícito es una característica central de Ruby: la última expresión evaluada en un método se convierte automáticamente en su valor de retorno. Por eso puedes escribir ``def doble(n); n * 2; end`` sin ``return`` y funciona igual. ``return`` solo es necesario cuando quieres salir del método antes de tiempo.

RESPUESTA 27: Falso - ``puts`` agrega un salto de línea al final de lo que imprime, mientras que ``print`` NO lo agrega. Por eso ``print "a"; print "b"`` muestra "ab" en la misma línea, pero ``puts "a"; puts "b"`` los muestra en líneas separadas. Además, ``puts`` convierte nil en una línea vacía y muestra los arreglos elemento por elemento.

RESPUESTA 28: Verdadero - Esa es exactamente la idea de los mixins: un módulo agrupa métodos que varias clases pueden compartir con ``include``, sin necesidad de que hereden de una clase común. Por ejemplo, el módulo ``Enumerable`` le da a distintas clases (Array, Hash, Range) métodos como ``map`` y ``select`` sin herencia entre ellas.

RESPUESTA 29: Verdadero - La interpolación ``#{}`` solo funciona en strings con comillas dobles. En un string con comillas simples, ``Hola #{nombre}`` se imprime literalmente con los caracteres ``#``, ``{``, etc. Regla práctica: comillas simples para texto literal, comillas dobles cuando necesitas interpolación o secuencias de escape como ``\n``. Excepción que confirma la regla: los símbolos con interpolación (``:"hola_#{nombre}"``) y las regex (``/patrón_#{var}/``) también usan comillas dobles por dentro, así que ahí la interpolación sí funciona.

RESPUESTA 30: Verdadero - ``super`` llama al método del mismo nombre en la clase padre (o en el siguiente ancestro de la cadena). Sin paréntesis, pasa automáticamente los mismos argumentos que recibió el método actual; con paréntesis y argumentos específicos (``super(arg)``), pasa esos. Es la herramienta clave para extender (no solo reemplazar) el comportamiento heredado.

RESPUESTA 31: Verdadero - Las gemas son la forma estándar de distribuir y reutilizar código Ruby: bibliotecas empaquetadas que instalas con ``gem install nombre_gema`` y gestionas con Bundler en proyectos grandes. Sitios como rubygems.org albergan miles de gemas gratuitas, desde Rails hasta herramientas para procesar imágenes o conectarte a APIs.

RESPUESTA 32: Falso - El símbolo ``^`` en una expresión regular significa "inicio del string", así que ``/^hola/`` solo coincide si el string EMPIEZA con "hola". Un string como "di hola amigo" no coincidiría porque "hola" no está al inicio. Para buscar "hola" en cualquier posición bastaría con ``/hola/`` sin el ``^``.

RESPUESTA 33: Falso - Las variables de instancia (``@nombre``) están encapsuladas dentro del objeto: desde fuera NO puedes leerlas ni escribirlas directamente. Para exponerlas necesitas métodos lectores/escritores, normalmente generados con ``attr_reader``, ``attr_writer`` o ``attr_accessor``. Intentar ``objeto.@nombre`` es un error de sintaxis; debes usar ``objeto.nombre`` si existe el lector. Este encapsulamiento es intencional: te obliga a controlar cómo se accede a los datos del objeto, lo que evita que código externo los deje en un estado inconsistente.

RESPUESTA 34: Verdadero - El bloque ``ensure`` se ejecuta siempre: si todo salió bien, si hubo una excepción atrapada por ``rescue``, e incluso si la excepción no fue atrapada. Por eso se usa para limpieza garantizada: cerrar archivos, cerrar conexiones a bases de datos o liberar recursos, pase lo que pase.

RESPUESTA 35: Falso - `5.times`` sí ejecuta el bloque 5 veces, pero los valores que pasa a la variable del bloque empiezan en 0, no en 1: son 0, 1, 2, 3 y 4. Si necesitas los números del 1 al 5, usa `(1..5).each`` en su lugar. Este desfase es una de las trampas clásicas para principiantes.

RESPUESTA 36: Solución:

```
def par_o_impar(numero)
  if numero % 2 == 0
    "par"
  else
    "impar"
  end
end

puts par_o_impar(4) # imprime "par"
puts par_o_impar(7) # imprime "impar"
```

Explicación: el operador módulo ``%`` devuelve el residuo de la división. Si ``numero % 2`` es 0, el número es par; si es 1, es impar. El método aprovecha el retorno implícito: la última expresión del ``if`` (el string "par" o "impar") se convierte en el valor de retorno sin necesidad de ``return``.

RESPUESTA 37: Solución:

```
contador = 1
while contador <= 10
  puts contador
  contador += 1
end
```

Explicación: el bucle ``while`` repite el bloque mientras la condición sea verdadera. En cada vuelta imprime el contador y luego lo incrementa en 1. El detalle crítico es el ``contador += 1``: sin esa línea el bucle sería infinito porque la condición nunca dejaría de cumplirse. El bucle termina cuando contador llega a 11 y ``11 <= 10`` es falso.

RESPUESTA 38: La salida exacta es:

```
0: ROJO
1: VERDE
2: AZUL
```


Explicación: `each_with_index` ejecuta el bloque por cada elemento pasando dos valores: el elemento (`color`) y su índice (`i`), que empieza en 0. Dentro del bloque, `color.upcase` convierte el string a mayúsculas y la interpolación `"#{i}: #{color.upcase}"` arma el texto final. Fíjate que el índice va antes del color y que todo el color queda en mayúsculas.

RESPUESTA 39: Solución:

```
def promedio(numeros)
  return 0 if numeros.empty?
  suma = 0
  numeros.each { |n| suma += n }
  suma.to_f / numeros.length
end

puts promedio([10, 20, 30]) # imprime 20.0
puts promedio([])          # imprime 0
```

Explicación: primero se protege el caso del arreglo vacío (dividir entre cero lanzaría un error), devolviendo 0 de inmediato. Luego se suma cada elemento con `each` y se divide entre la cantidad de elementos. Se usa `suma.to_f` para convertir a decimal antes de dividir: así `promedio([1, 2])` devuelve 1.5 en lugar de 1 (recuerda que la división entre enteros trunca en Ruby).

RESPUESTA 40: El error es que el método `saluda` requiere un argumento (`nombre`), pero se le llama sin argumentos: `saluda()`. Ruby lanza `ArgumentError: wrong number of arguments (given 0, expected 1)`. Versión corregida:

```
def saluda(nombre)
  puts "Hola " + nombre
end

saluda("Ana")
```

Explicación: en Ruby el número de argumentos debe coincidir con el número de parámetros, a menos que uses valores por defecto o argumentos opcionales. La corrección más directa es pasarle el nombre al llamar al método. Alternativa válida: darle un valor por defecto al parámetro con `def saluda(nombre = "mundo")`, y entonces `saluda()` sí funcionaría.

RESPUESTA 41: Solución:

```

class Rectangulo
  def initialize(base, altura)
    @base = base
    @altura = altura
  end

  def area
    @base * @altura
  end

  def perimetro
    2 * @base + 2 * @altura
  end
end

rect = Rectangulo.new(5, 3)
puts rect.area # imprime 15

```

Explicación: `initialize` es el constructor que se ejecuta automáticamente al llamar `Rectangulo.new(5, 3)`; guarda los valores en las variables de instancia `@base` y `@altura` para que estén disponibles en todos los métodos del objeto. `area` multiplica base por altura ($5 * 3 = 15$) y `perimetro` suma dos veces cada lado. El retorno es implícito en ambos métodos.

RESPUESTA 42: La salida exacta es:

```

cero
10

```

Explicación: `misterio(0)` entra al método con $n = 0$, la guarda `return "cero" if n == 0` se cumple y el método sale de inmediato devolviendo "cero" (el `return` explícito interrumpe antes de llegar a `n \* 2`). `misterio(5)` no cumple la guarda, así que llega a la última línea y devuelve `5 \* 2 = 10` por retorno implícito.

RESPUESTA 43: Solución:

```

palabra = "programacion"
vocales = "aeiou"
conteo = {}

palabra.each_char do |letra|
  if vocales.include?(letra)
    conteo[letra] = (conteo[letra] || 0) + 1
  end
end

puts conteo

```

Explicación: `each_char` recorre la palabra letra por letra. Si la letra es una vocal, se incrementa su contador en el hash. El truco está en `(conteo[letra] || 0)`: la primera vez que aparece una vocal, `conteo[letra]` es `nil`, así que se usa 0 como base y queda en 1; las siguientes veces suma sobre el valor ya guardado. La salida es `{"o"=>2, "a"=>2, "i"=>1}` (el orden de las claves refleja el orden de aparición: o, a, i).

RESPUESTA 44: Solución:

```

puts "Escribe tu edad:"
entrada = gets.chomp

begin
  edad = Integer(entrada)
  puts "En 10 años tendrás #{edad + 10}"
rescue ArgumentError
  puts "Eso no es un número válido. Inténtalo de nuevo."
end

```

Explicación: el problema del código original es que `"abc".to_i` devuelve 0 sin avisar, así que el programa seguiría como si la edad fuera 0. La corrección usa `Integer(entrada)`, que SÍ lanza `ArgumentError` cuando el texto no es un número válido. Ese error se atrapa con `begin/rescue` y se muestra un mensaje amable en lugar de dejar que el programa se rompa o calcule con datos basura. Esta es la forma correcta de validar entrada numérica en Ruby. Alternativa válida: usar una expresión regular como `entrada.match?(/^\d+$/)` para verificar que solo haya dígitos antes de convertir; esa técnica la viste en la Unidad 9.

RESPUESTA 45: Solución:

```

class Vehiculo
  def describe
    "Soy un vehículo"
  end
end

class Coche < Vehiculo
  def describe
    super + " y soy un coche"
  end
end

coche = Coche.new
puts coche.describe # imprime "Soy un vehículo y soy un coche"

```

Explicación: `Coche < Vehiculo` declara la herencia. El método `describe` del padre DEVUELVE el string (no lo imprime) para que el hijo pueda reutilizarlo. En el hijo, `super` llama al `describe` del padre y devuelve "Soy un vehículo"; luego se le concatena " y soy un coche" con el operador `+`. El resultado final es "Soy un vehículo y soy un coche". La clave del ejercicio era que el padre devuelva en lugar de imprimir, porque `puts` devuelve `nil` y no se podría concatenar.

RESPUESTA 46: La salida exacta es:

```

R*by *s g*n**l
3

```

Explicación: en la primera línea, `gsub(/[aeiou]/, "\*")` reemplaza CADA vocal (minúscula) por un asterisco: "Ruby" pierde la "u" (R*by), "es" pierde la "e" (*s), "genial" pierde la "e", la "i" y la "a" (g*n!). **Ojo: la "R" mayúscula NO se reemplaza porque el patrón solo incluye vocales minúsculas.** En la segunda línea, `split(" ")` divide el string ORIGINAL (gsub no modifica el original, devuelve uno nuevo) en ["Ruby", "es", "genial"], y `.length` cuenta 3 elementos.

RESPUESTA 47: Solución:

```

module Volador
  def volar
    puts "Estoy volando"
  end
end

class Pajaro
  include Volador
end

piolin = Pajaro.new
piolin.volar # imprime "Estoy volando"

```

Explicación: el módulo `Volador` agrupa el método `volar`. Con `include Volador` dentro de la clase, ese método se mezcla (mixin) en `Pajaro` como método de instancia, sin necesidad de herencia. Así, cualquier objeto `Pajaro.new` puede llamar a `volar`. Esta es la forma idiomática de compartir comportamiento entre clases no relacionadas en Ruby. Bonus: los módulos también sirven como espacios de nombres (namespaces) para organizar código, por ejemplo `module MiApp; class Usuario; end; end` evita choques de nombres con otras clases `Usuario`.

RESPUESTA 48: Solución:

```

begin
  File.open("notas.txt", "r") do |archivo|
    archivo.each_line do |linea|
      puts linea if linea.downcase.include?("ruby")
    end
  end
rescue Errno::ENOENT
  puts "Archivo no encontrado"
end

```

Explicación: `File.open` con bloque abre el archivo en modo lectura ("r") y lo cierra automáticamente. `each_line` recorre línea por línea sin cargar todo el archivo en memoria (importante si el archivo es grande). Para que la búsqueda no distinga mayúsculas de minúsculas, se convierte cada línea a minúsculas con `downcase` antes de buscar "ruby" con `include?`. Si el archivo no existe, Ruby lanza `Errno::ENOENT`, que se atrapa con `rescue` para imprimir el mensaje amable en lugar de mostrar un error técnico.

RESPUESTA 49: La salida exacta es:

Explicación: vamos paso a paso. ``select { |n| n.odd? }` filtra el arreglo quedándose solo con los impares: [1, 3, 5] (el 2 y el 4 se descartan). Luego ``each`` recorre esos tres números y los va sumando en la variable ``suma``:  $0 + 1 = 1$ ,  $1 + 3 = 4$ ,  $4 + 5 = 9$ . Finalmente ``puts suma`` imprime 9. Fíjate que ``select`` devuelve un arreglo nuevo y que la variable ``suma`` definida fuera del bloque sí puede modificarse dentro de él (los bloques ven las variables locales del entorno donde se crean).

RESPUESTA 50: Solución:

```
def fizzbuzz(n)
  (1..n).each do |i|
    if i % 3 == 0 && i % 5 == 0
      puts "FizzBuzz"
    elsif i % 3 == 0
      puts "Fizz"
    elsif i % 5 == 0
      puts "Buzz"
    else
      puts i
    end
  end
end

fizzbuzz(15)
```

Explicación: ``(1..n).each`` recorre los números del 1 al n. El operador ``%`` detecta los múltiplos: si el residuo de dividir entre 3 es 0, es múltiplo de 3. El orden de las condiciones es CRÍTICO: primero se revisa el caso de múltiplo de ambos (15, 30...), porque si revisaras el de 3 primero, el 15 imprimiría "Fizz" y nunca llegaría a "FizzBuzz". Con ``fizzbuzz(15)``, la salida es: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz. Los dos errores más comunes aquí son poner las condiciones en el orden equivocado y confundir qué palabra va con cada múltiplo, así que siempre prueba tu código con un ejemplo como el 15.

Cómo calificar tu examen y qué sigue

Ya terminaste las 50 preguntas. Ahora califícate con honestidad: cada pregunta vale 2 puntos, el examen vale 100. Suma tus aciertos, multiplícalos por 2 y ubica tu resultado:

90 a 100: Excelente. Dominas los fundamentos de Ruby. Estás listo para dar el salto a Ruby on Rails o para construir tus propios proyectos. Repasa solo los temas donde fallaste y sigue adelante.

70 a 88: Bien. Tienes una base sólida, pero hay huecos. Identifica las preguntas que fallaste, ubica a qué unidad pertenece cada tema y repasa esas unidades antes de seguir. No avances con dudas arrastradas.

50 a 68: En proceso. Entendiste una parte importante, pero te falta práctica. Vuelve a las unidades donde más fallaste, haz de nuevo los ejercicios y reintenta el examen en una o dos semanas. No es un fracaso: es un diagnóstico.

Menos de 50: Vuelve al inicio sin pena. Relee el libro con más calma, escribe cada ejemplo a mano y haz todos los ejercicios. La programación se aprende programando, no leyendo. Date otra vuelta completa y vuelve al examen.

Un consejo importante: no te quedes solo con la calificación. Por cada pregunta que fallaste, escribe en un cuaderno qué concepto no dominabas y un ejemplo de código que lo demuestre. Esa lista de errores es oro: es tu plan de repaso personalizado.

¿Y después del examen? Tienes varios caminos, todos buenos:

Si te gustó el desarrollo web, sigue con Ruby on Rails. Ya tienes la base del lenguaje, que es lo más difícil; Rails te va a parecer natural. En lucasegaacademy.com encontrarás el curso de Rails para continuar.

Si te gustó automatizar, profundiza en scripting: aprende a trabajar con APIs, a procesar archivos CSV y JSON, y a programar tareas automáticas. Ruby brilla en eso.

Si quieres ser mejor programador en general, aprende a escribir pruebas automatizadas (RSpec y Minitest son los estándares de Ruby) y estudia estructuras de datos y algoritmos usando Ruby como tu lenguaje.

Un consejo extra que vale oro: arma un portafolio con 3 o 4 proyectos pequeños pero bien hechos. No tiene que ser nada gigantesco: un script que organice tus archivos por tipo, un juego de adivinanzas en la terminal con puntajes guardados en un archivo, un convertidor de monedas que lea datos de un CSV, un blog simple con Rails. Lo importante es que cada proyecto use varias de las cosas que aprendiste (clases, archivos, excepciones, regex) y que el código esté limpio y comentado. Cuando busques trabajo o clientes, ese portafolio habla más fuerte que cualquier certificado.

Y si te animas, contribuye a un proyecto de código abierto en Ruby, aunque sea corrigiendo documentación o traduciendo algo al español. Es la forma más rápida de leer código de programadores experimentados y de aprender cómo se trabaja en equipo con Git.

Y hagas lo que hagas, sigue programando todos los días. El examen fue la meta de este libro, pero en programación nunca se deja de aprender: cada proyecto te enseña algo nuevo. Lo importante es que ya tienes la base, las herramientas y el hábito. Lo demás es seguir construyendo.

Gracias por llegar hasta aquí. Nos vemos en el siguiente curso de Lucas EGA Academy. Y recuerda: el mejor programador no es el que nunca se equivoca, sino el que nunca deja de practicar.

Las 10 trampas más comunes de este examen (y cómo evitarlas)

Si fallaste varias preguntas, es probable que hayas caído en alguna de estas trampas clásicas. Vale la pena conocerlas porque aparecen una y otra vez, no solo en exámenes sino en código real:

1. La división entre enteros trunca. `7 / 2` es 3, no 3.5. Si necesitas decimales, convierte uno de los operandos con `.to_f` antes de dividir.
2. `times` empieza en 0. `5.times` pasa 0, 1, 2, 3, 4 al bloque. Si quieres del 1 al 5, usa `(1..5).each`.
3. Los índices empiezan en 0. `frutas[1]` es el SEGUNDO elemento. El primero es `frutas[0]`.
4. `each` no transforma nada. Solo itera y devuelve el arreglo original. Para transformar usa `map`; para filtrar usa `select`.
5. `map` no modifica el original. Devuelve un arreglo nuevo. El que modifica es `map!` (con admiración).
6. Solo `false` y `nil` son falsos. El 0, el `""` y el `[]` son verdaderos en Ruby. Esto sorprende a todos los que vienen de otros lenguajes.
7. La interpolación necesita comillas dobles. `"Hola #{nombre}"` imprime el texto literal; `"\"#{nombre}\""` sustituye el valor.
8. `puts` agrega salto de línea, `print` no. Parece obvio, pero en preguntas de salida exacta es la diferencia entre acertar y fallar.
9. En regex, `^` es inicio del string. `/^hola/` no encuentra "hola" en medio de un texto, solo al principio.

10. El orden de las condiciones importa. En FizzBuzz (y en cualquier ``if/elsif``), el caso más específico (múltiplo de 3 Y de 5) debe revisarse ANTES que los casos generales, o nunca se alcanzará.

Domina estas diez y habrás eliminado la mayoría de los errores "tontos" que hacen perder puntos. No son tontos en realidad: son los detalles que separan a quien leyó el libro de quien lo entendió.

Curso Completo de Ruby

Libro de programación de Lucas EGA Academy: explicaciones extensas unidad por unidad, práctica guiada y examen final para dominar el lenguaje desde cero.

Colibrí Editorial EGA

lucasegaacademy.com