Curso de Ruby on Rails

Diego Saavedra

Jun 2, 2024

Table of contents

1 Bienvenido **6**
- 1.1 ¿De qué trata este curso? . . . 6
- 1.2 ¿Para quién es este curso? . . . 6
- 1.3 ¿Cómo contribuir? . . . 6

I Unidad 1: Fundamentos de Ruby **8**

2 Introducción **9**

3 Instalación **10**
- 3.1 Windows . . . 10
- 3.2 Gnu/Linux . . . 10
- 3.3 Configuración de RVM o rbenv para la gestión de versiones. . . . 10
  - 3.3.1 RVM . . . 11
  - 3.3.2 rbenv . . . 11
- 3.4 Introducción al Intérprete IRB y Archivos Ruby . . . 12
- 3.5 Creación y ejecución de scripts Ruby desde la terminal. . . . 12
- 3.6 Ejercicios Prácticos . . . 12
- 3.7 Conclusiones . . . 14

II Unidad 2: Estructuras Básicas y Tipos de Datos **15**

4 Variables y Tipos de Datos **16**
- 4.1 Introducción . . . 16
- 4.2 Convenciones y Tipos de Datos en Ruby . . . 16
- 4.3 Tipos de Variables . . . 18
  - 4.3.1 Variables Locales . . . 18
  - 4.3.2 Variables Globales . . . 18
  - 4.3.3 Variables de Instancia . . . 18
  - 4.3.4 Variables de Clase . . . 19
  - 4.3.5 Tipos de Datos . . . 19
- 4.4 Operadores y Expresiones . . . 22
  - 4.4.1 Operadores Aritméticos . . . 22
  - 4.4.2 Operadores de Comparación . . . 23
  - 4.4.3 Operadores Lógicos . . . 25
  - 4.4.4 Ejercicios Prácticos . . . 25
  - 4.4.5 Conclusiones . . . 26

III Unidad 3: Métodos y Bloques **27**

5 Definición y Uso de Métodos **28**
5.1 Introducción . . . 28
5.1.1 Definición de Métodos . . . 28
5.1.2 Uso de Métodos . . . 28
5.1.3 Valores de Retorno . . . 28
5.2 Bloques, Procs, y Lambdas . . . 29
5.2.1 Bloques . . . 29
5.2.2 Procs . . . 29
5.2.3 Lambdas . . . 30
5.2.4 Diferencias entre Procs y Lambdas . . . 30
5.3 Creación y Uso de Métodos . . . 30
5.4 Métodos de Ruby . . . 31
5.5 Ejercicios Prácticos . . . 33
5.6 Conclusiones . . . 37

IV Unidad 4: Programación Orientada a Objetos **38**

6 Clases y Objetos **39**
6.1 Introducción . . . 39
6.2 Conceptos básicos de la POO . . . 39
6.3 Creación de Clases y Objetos . . . 39
6.4 Atributos y Métodos de Instancia . . . 40

7 Ejemplos Prácticos **41**
7.1 Ejemplo 1: Crear una clase y un objeto . . . 41
7.2 Ejemplo 2: Crear una clase con herencia . . . 41
7.3 Ejemplo 3: Crear un módulo y compartir comportamiento . . . 42

8 Ejercicios Prácticos **44**
8.0.1 Herencia y Polimorfismo . . . 44
8.0.2 Polimorfismo . . . 45
8.0.3 Módulos y Mixins . . . 46
8.0.4 Ejercicios Prácticos . . . 46

9 Conclusiones **48**

V Unidad 5: Manejo de Errores y Pruebas **49**

10 Manejo de Excepciones **50**
10.1 Introducción . . . 50
10.2 Rescate de Excepciones . . . 50
10.3 Creación de Excepciones Personalizadas . . . 51

11 Conclusiones **56**

VI Unidad 6: Ruby Avanzado **57**

12 Metaprogramación y Enumerables en Ruby **58**
12.1 Introducción . . . 58
12.2 Metaprogramación . . . 58
12.3 Enumerables y Enumeradores . . . 60
12.4 Ejercicios Prácticos . . . 61
12.5 Conclusiones . . . 64

VII Unidad 7: Ruby y la Web **65**

13 Introducción a Ruby on Rails **66**
13.1 ¿Qué es Ruby on Rails? . . . 66
13.2 Instalación de Ruby on Rails . . . 66

14 Creación de una nueva aplicación Rails **68**
14.1 Parte 1: Configuración del Proyecto . . . 68
14.1.1 Creación del Proyecto: . . . 68
14.1.2 Navega a la Carpeta del Proyecto: . . . 68
14.1.3 Inicia el Servidor: . . . 68
14.2 Parte 2: Generación del Modelo . . . 69
14.2.1 Conceptos Básicos . . . 69
14.2.2 Generación del Modelo: . . . 69
14.2.3 Ejecutar la Migración: . . . 69
14.3 Parte 3: Generación del Controlador . . . 69
14.3.1 Conceptos Básicos . . . 69
14.3.2 Generación del Controlador: . . . 69
14.4 Parte 4: Rutas . . . 70
14.4.1 Conceptos Básicos . . . 70
14.4.2 Configurar las Rutas: . . . 70
14.5 Parte 5: Vistas . . . 70
14.5.1 Conceptos Básicos . . . 70
14.5.2 Crear las Vistas: . . . 70
14.5.3 Controlador y Acción index: . . . 71
14.6 Parte 6: CRUD - Create . . . 71
14.6.1 Conceptos Básicos . . . 71
14.6.2 Formulario para Crear un Nuevo Ítem: . . . 71
14.6.3 Acción new y create en el Controlador: . . . 71
14.7 Parte 7: CRUD - Read . . . 72
14.7.1 Conceptos Básicos . . . 72
14.7.2 Vista show: . . . 72
14.8 Parte 8: CRUD - Update . . . 73
14.8.1 Conceptos Básicos . . . 73
14.8.2 Formulario para Editar un Ítem: . . . 73
14.8.3 Acción edit y update en el Controlador: . . . 73
14.8.4 Parte 9: CRUD - Delete . . . 74
14.8.5 Conceptos Básicos . . . 74
14.8.6 Acción destroy en el Controlador: . . . 74

14.9 Parte 10: Pruebas y Verificación 76
14.9.1 Conceptos Básicos 76
14.9.2 Verificar la Funcionalidad: 76
14.10Pruebas Manuales: 76
14.11Parte 11: Estilo y Mejoras 77
14.11.1 Conceptos Básicos 77
14.11.2 Agregar Estilo: 77

15 Actividad **79**

16 Conclusión **83**

1 Bienvenido

¡Bienvenido al Curso Completo de Ruby on Rails!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso completo me llevará desde los fundamentos básicos de la programación hasta la construcción de aplicaciones prácticas utilizando el Framework Rails.

A través de una combinación de teoría y ejercicios prácticos, me sumergiré en los conceptos esenciales del desarrollo web y avanzaré hacia la creación de proyectos del mundo real.

Desde la configuración del entorno de desarrollo hasta la construcción de una aplicación web de pila completa, este curso me proporcionará una comprensión sólida y experiencia práctica con Ruby on Rails.

1.2 ¿Para quién es este curso?

Este curso está diseñado para principiantes y aquellos con poca o ninguna experiencia en programación.

Ya sea que sea un estudiante curioso, un profesional que busca cambiar de carrera o simplemente alguien que quiere aprender desarrollo web, este curso es para usted. Desde adolescentes hasta adultos, todos son bienvenidos a participar y explorar el emocionante mundo del desarrollo web con Ruby on Rails.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en linea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

Esperamos que disfrute este emocionante viaje de aprendizaje y descubrimiento en el mundo del desarrollo web con Django y React!

Part I

Unidad 1: Fundamentos de Ruby

2 Introducción

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro Matsumoto, quien comenzó a trabajar en Ruby en 1993 y lo presentó públicamente en 1995. Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidades con otros lenguajes de programación como Lisp, Lua, Dylan y CLU.

3 Instalación

Para realizar la instalación de Ruby en diferentes sistemas operativos, se recomienda seguir las instrucciones oficiales de la documentación de Ruby.

3.1 Windows

Para instalar Ruby en Windows, se recomienda descargar el instalador de RubyInstaller desde la página oficial de RubyInstaller.

1. Descargar el instalador de RubyInstaller desde la página oficial de RubyInstaller.
2. Ejecutar el instalador de RubyInstaller y seguir las instrucciones del asistente de instalación.
3. Verificar la instalación de Ruby ejecutando el comando **ruby -v** en la terminal.

```
ruby -v
```

3.2 Gnu/Linux

Para instalar Ruby en Gnu/Linux, se recomienda seguir las instrucciones de la documentación oficial de Ruby.

1. Instalar Ruby utilizando el gestor de paquetes de la distribución de Gnu/Linux.
2. Verificar la instalación de Ruby ejecutando el comando **ruby -v** en la terminal.

```
ruby -v
```

3.3 Configuración de RVM o rbenv para la gestión de versiones.

Para la gestión de versiones de Ruby, se recomienda utilizar RVM o rbenv.

3.3.1 RVM

RVM (Ruby Version Manager) es un gestor de versiones de Ruby que permite instalar y gestionar múltiples versiones de Ruby en un mismo sistema.

Para instalar RVM, se recomienda seguir las instrucciones de la documentación oficial de RVM.

1. Instalar RVM utilizando el script de instalación de RVM.

```
\curl -sSL https://get.rvm.io | bash -s stable
```

2. Cargar RVM en la terminal.

```
source ~/.rvm/scripts/rvm
```

3. Instalar una versión de Ruby utilizando RVM.

```
rvm install ruby
```

4. Verificar la instalación de Ruby ejecutando el comando **ruby -v** en la terminal.

```
ruby -v
```

3.3.2 rbenv

rbenv es un gestor de versiones de Ruby que permite instalar y gestionar múltiples versiones de Ruby en un mismo sistema.

Para instalar rbenv, se recomienda seguir las instrucciones de la documentación oficial de rbenv.

1. Instalar rbenv utilizando el script de instalación de rbenv.

```
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

2. Añadir rbenv al PATH del sistema.

```
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
```

3. Inicializar rbenv en la terminal.

```
eval "$(rbenv init -)"
```

3.4 Introducción al Intérprete IRB y Archivos Ruby

El Intérprete IRB (Interactive Ruby) es una herramienta que permite ejecutar comandos Ruby de forma interactiva en la terminal. Para acceder a IRB, simplemente abre una terminal y escribe **irb**.

En IRB, puedes ejecutar cualquier comando Ruby y ver los resultados de inmediato. Por ejemplo, puedes probar operaciones matemáticas:

```
2 + 2
```

3.5 Creación y ejecución de scripts Ruby desde la terminal.

Para crear y ejecutar scripts Ruby desde la terminal, simplemente crea un archivo con extensión **.rb** y escribe tu código Ruby en el archivo. Luego, puedes ejecutar el script Ruby utilizando el comando **ruby** seguido del nombre del archivo.

Por ejemplo, crea un archivo llamado **hello.rb** con el siguiente contenido:

```
puts "Hello, World!"
```

Luego, ejecuta el script Ruby utilizando el comando **ruby**:

```
ruby hello.rb
```

3.6 Ejercicios Prácticos

1. Ejercicio 1: Saludo personalizado

Escribe un programa que solicite al usuario su nombre y luego imprima un saludo personalizado.

Ver solución

```
puts "Ingrese su nombre:"
nombre = gets.chomp
puts "¡Hola, #{nombre}! Bienvenido a Ruby."
```

En el código anterior, utilizamos el método **gets.chomp** para obtener la entrada del usuario y eliminar el salto de línea. Luego, interpolamos la variable **nombre** en el mensaje de saludo utilizando la sintaxis **#{nombre}**.

2. Ejercicio 2: Suma de dos números

Escribe un programa que solicite al usuario dos números enteros y luego imprima la suma de los dos números.

Ver solución

```
puts "Ingrese el primer número:"
num1 = gets.chomp.to_i

puts "Ingrese el segundo número:"
num2 = gets.chomp.to_i

suma = num1 + num2

puts "La suma de #{num1} y #{num2} es igual a #{suma}."
```

En el código anterior, utilizamos el método **gets.chomp.to_i** para obtener la entrada del usuario y convertirla a un número entero. Luego, calculamos la suma de los dos números y la imprimimos en la pantalla.

3. Ejercicio 3: Cálculo del área de un triángulo

Escribe un programa que solicite al usuario la base y la altura de un triángulo y luego calcule y imprima el área del triángulo.

Ver solución

```
puts "Ingrese la base del triángulo:"
base = gets.chomp.to_f

puts "Ingrese la altura del triángulo:"
altura = gets.chomp.to_f

area = 0.5 * base * altura

puts "El área del triángulo es igual a #{area}."
```

En el código anterior, utilizamos el método **gets.chomp.to_f** para obtener la entrada del usuario y convertirla a un número de coma flotante. Luego, calculamos el área del triángulo utilizando la fórmula **0.5 * base * altura** y la imprimimos en la pantalla.

4. Ejercicio 4: Conversión de temperatura

Escribe un programa que solicite al usuario una temperatura en grados Celsius y luego la convierta a grados Fahrenheit.

Ver solución

```ruby
puts "Ingrese la temperatura en grados Celsius:"

celsius = gets.chomp.to_f
fahrenheit = (celsius * 9/5) + 32

puts "#{celsius} grados Celsius equivale a #{fahrenheit} grados Fahrenheit."
```

En el código anterior, utilizamos el método **gets.chomp.to_f** para obtener la entrada del usuario y convertirla a un número de coma flotante. Luego, calculamos la temperatura en grados Fahrenheit utilizando la fórmula **(celsius * 9/5) + 32** y la imprimimos en la pantalla.

5. Ejercicio 5: Números pares e impares

Escribe un programa que solicite al usuario un número entero y determine si es par o impar.

Ver solución

```ruby
puts "Ingrese un número entero:"
numero = gets.chomp.to_i

if numero % 2 == 0
puts "#{numero} es un número par."
else
puts "#{numero} es un número impar."
end
```

En el código anterior, utilizamos el operador módulo % para determinar si el número es par o impar. Si el resto de la división por 2 es igual a 0, el número es par; de lo contrario, es impar.

3.7 Conclusiones

Ruby es un lenguaje de programación versátil y potente que se utiliza en una amplia variedad de aplicaciones, desde desarrollo web hasta scripting y automatización. Con una sintaxis elegante y fácil de leer, Ruby es un excelente lenguaje para principiantes y expertos por igual. Con la instalación de Ruby y la práctica de ejercicios prácticos, puedes comenzar a explorar y aprender Ruby de forma efectiva.

¡Buena suerte en tu viaje de aprendizaje de Ruby!

Part II

Unidad 2: Estructuras Básicas y Tipos de Datos

4 Variables y Tipos de Datos

En este capítulo vamos a ver cómo se declaran variables en Python y los tipos de datos que podemos almacenar en ellas.

4.1 Introducción

Las variables son espacios de memoria que se utilizan para almacenar valores. En Ruby, las variables se declaran de la siguiente manera:

```
variable = valor
```

Donde **variable** es el nombre de la variable y **valor** es el valor que se le asigna.

4.2 Convenciones y Tipos de Datos en Ruby

Es recomendable tomar en cuenta algunas convenciones al momento de declarar variables en Ruby:

1. Los nombres de las variables deben comenzar con una letra minúscula o un guión bajo.

Ejemplo:

```
_nombre = "Juan"
apellido = "Perez"
puts _nombre # Juan
puts apellido # Perez
```

En el ejemplo anterior, la variable **_nombre** almacena el valor “Juan” y la variable **apellido** almacena el valor “Perez”.

2. Los nombres de las variables deben ser descriptivos y significativos.

Ejemplo:

```ruby
nombre_completo = "Juan Perez"
puts nombre_completo # Juan Perez

a = "Juan" # Esto según la convensión no es correcto
b = 27 # Esto según la convensión no es correcto

puts a # Juan
puts b # 27
```

En el ejemplo anterior, la variable **nombre_completo** almacena el valor "Juan Perez", sin embargo, las variables **a** y **b** no son descriptivas ni significativas.

3. Los nombres de las variables deben estar en minúsculas y separados por guiones bajos.

Ejemplo:

```ruby
nombre_completo = "Juan Perez"
puts nombre_completo # Juan Perez
```

En el ejemplo anterior, la variable **nombre_completo** almacena el valor "Juan Perez".

4. Los nombres de las variables no deben comenzar con un número.

Ejemplo:

```ruby
nombre = "Juan"
puts nombre # Juan

4nombre = "Juan" # Esto no es correcto
puts 4nombre # Juan
```

En el ejemplo anterior, la variable **nombre** almacena el valor "Juan", sin embargo, la variable **4nombre** no es válida porque comienza con un número.

5. Los nombres de las variables no deben ser palabras reservadas.

Ejemplo:

```ruby
nombre = "Juan"
puts nombre # Juan

def = "Juan" # Esto no es correcto
puts def # Juan
```

En el ejemplo anterior, la variable **nombre** almacena el valor "Juan", sin embargo, la variable **def** no es válida porque es una palabra reservada.

4.3 Tipos de Variables

En Ruby existen diferentes tipos de variables,

1. variables locales,
2. Variables globales,
3. Variables de instancia y
4. Variables de clase.

4.3.1 Variables Locales

Las variables locales se declaran con un nombre que comienza con una letra minúscula o un guión bajo. Por ejemplo:

```
nombre = "Juan"
puts nombre # Juan
puts defined?(nombre) # local-variable
```

En el ejemplo anterior, la variable **nombre** almacena el valor "Juan". La función **defined?** nos permite saber si una variable está definida y en este caso nos devuelve **local-variable**.

4.3.2 Variables Globales

Las variables globales se declaran con un signo de dólar ($) seguido de un nombre. Por ejemplo:

```
$nombre = "Juan"
puts $nombre # Juan
puts defined?($nombre) # global-variable
```

En el ejemplo anterior, la variable **$nombre** almacena el valor "Juan". La función **defined?** nos permite saber si una variable está definida y en este caso nos devuelve **global-variable**.

4.3.3 Variables de Instancia

Las variables de instancia se declaran con un signo de arroba (@) seguido de un nombre. Por ejemplo:

```
@nombre = "Juan"
puts @nombre # Juan
puts defined?(@nombre) # instance-variable
```

En el ejemplo anterior, la variable **(nombre?)** almacena el valor "Juan". La función **defined?** nos permite saber si una variable está definida y en este caso nos devuelve **instance-variable**.

4.3.4 Variables de Clase

Las variables de clase se declaran con dos signos de arroba (@@) seguidos de un nombre. Por ejemplo:

```
class HelloWorld
  @@nombre = "Juan"
end

puts HelloWorld.class_variable_get(:@@nombre) # Juan
puts defined?(HelloWorld.class_variable_get(:@@nombre)) # class-variabley
```

En el ejemplo anterior, la variable **@(nombre?)** almacena el valor "Juan". La función **class_variable_get** nos permite obtener el valor de una variable de clase y la función **defined?** nos permite saber si una variable está definida y en este caso nos devuelve **class-variable**.

4.3.5 Tipos de Datos

Ahora vamos a conocer los diferentes tipos de datos que existen en Ruby:

1. **Integers**: Números enteros. Por ejemplo: 1, 2, 3, 4, 5.
2. **Floats**: Números decimales. Por ejemplo: 1.5, 2.5, 3.5, 4.5, 5.5.
3. **Strings**: Cadenas de texto. Por ejemplo: "Hola Mundo".
4. **Symbols**: Símbolos. Por ejemplo: :hola_mundo.
5. **Arrays**: Listas de elementos. Por ejemplo: [1, 2, 3, 4, 5].
6. **Hashes**: Colecciones de pares clave-valor. Por ejemplo: {nombre: "Juan", edad: 25}.
7. **Booleans**: Valores booleanos. Por ejemplo: true, false.
8. **Nil**: Valor nulo. Por ejemplo: nil.
9. **Ranges**: Rangos de valores. Por ejemplo: 1..5.

4.3.5.1 Integers

Los números enteros son valores numéricos que no tienen parte decimal. En Ruby, los números enteros se representan con la clase Integer. Por ejemplo:

```
numero = 5
puts numero # 5
puts numero.class # Integer
```

En el ejemplo anterior, la variable **numero** almacena el valor 5, que es un número entero.

4.3.5.2 Floats

Los números decimales son valores numéricos que tienen parte decimal. En Ruby, los números decimales se representan con la clase Float. Por ejemplo:

```ruby
numero = 5.5
puts numero # 5.5
puts numero.class # Float
```

En el ejemplo anterior, la variable **numero** almacena el valor 5.5, que es un número decimal.

4.3.5.3 Strings

Las cadenas de texto son secuencias de caracteres que se utilizan para representar texto. En Ruby, las cadenas de texto se representan con la clase String. Por ejemplo:

```ruby
nombre = "Juan"
puts nombre # Juan
puts nombre.class # String
```

En el ejemplo anterior, la variable **nombre** almacena el valor "Juan", que es una cadena de texto.

4.3.5.4 Symbols

Los símbolos son valores inmutables que se utilizan para representar nombres o identificadores. En Ruby, los símbolos se representan con la clase Symbol. Por ejemplo:

```ruby
nombre = :juan
puts nombre # juan
puts nombre.class # Symbol
```

En el ejemplo anterior, la variable **nombre** almacena el valor :juan, que es un símbolo.

4.3.5.5 Arrays

Los arrays son listas de elementos que se utilizan para almacenar colecciones de valores. En Ruby, los arrays se representan con la clase Array. Por ejemplo:

```ruby
numeros = [1, 2, 3, 4, 5]
puts numeros # [1, 2, 3, 4, 5]
puts numeros.class # Array
```

En el ejemplo anterior, la variable **numeros** almacena un array con los valores 1, 2, 3, 4 y 5.

4.3.5.6 Hashes

Los hashes son colecciones de pares clave-valor que se utilizan para almacenar datos de forma estructurada. En Ruby, los hashes se representan con la clase Hash. Por ejemplo:

```ruby
datos = {nombre: "Juan", edad: 25}
puts datos # {:nombre=>"Juan", :edad=>25}
puts datos.class # Hash
```

En el ejemplo anterior, la variable **datos** almacena un hash con los pares clave-valor nombre: "Juan" y edad: 25.

4.3.5.7 Booleans

Los valores booleanos son valores lógicos que representan verdadero o falso. En Ruby, los valores booleanos se representan con las palabras reservadas true y false. Por ejemplo:

```ruby
verdadero = true
puts verdadero # true
puts verdadero.class # TrueClass

falso = false
puts falso # false
puts falso.class # FalseClass
```

En el ejemplo anterior, la variable **verdadero** almacena el valor true, que representa verdadero, y la variable **falso** almacena el valor false, que representa falso.

4.3.5.8 Nil

El valor nulo es un valor especial que representa la ausencia de un valor. En Ruby, el valor nulo se representa con la palabra reservada nil. Por ejemplo:

```ruby
valor = nil
puts valor # nil
puts valor.class # NilClass
```

En el ejemplo anterior, la variable **valor** almacena el valor nil, que representa la ausencia de un valor.

4.3.5.9 Ranges

Los rangos son secuencias de valores que se utilizan para representar un intervalo de valores. En Ruby, los rangos se representan con la clase Range. Por ejemplo:

```ruby
rango = 1..5
puts rango # 1..5
puts rango.class # Range
```

En el ejemplo anterior, la variable **rango** almacena un rango con los valores del 1 al 5.

4.4 Operadores y Expresiones

Ruby cuenta con una serie de operadores que nos permiten realizar operaciones aritméticas, lógicas y de comparación. A continuación, vamos a ver algunos de los operadores más comunes en Ruby:

4.4.1 Operadores Aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas. A continuación, se muestran algunos de los operadores aritméticos más comunes en Ruby:

1. +: Suma dos valores.

Ejemplo:

```ruby
resultado = 1 + 2
puts resultado # 3
```

En el ejemplo anterior, la variable **resultado** almacena el valor 3, que es el resultado de sumar 1 y 2.

2. -: Resta dos valores.

Ejemplo:

```ruby
resultado = 2 - 1
puts resultado # 1
```

En el ejemplo anterior, la variable **resultado** almacena el valor 1, que es el resultado de restar 1 de 2.

3. *: Multiplica dos valores.

Ejemplo:

```ruby
resultado = 2 * 3
puts resultado # 6
```

En el ejemplo anterior, la variable **resultado** almacena el valor 6, que es el resultado de multiplicar 2 por 3.

4. /: Divide dos valores.

Ejemplo:

```
resultado = 6 / 2
puts resultado # 3
```

En el ejemplo anterior, la variable **resultado** almacena el valor 3, que es el resultado de dividir 6 entre 2.

5. %: Obtiene el residuo de una división.

Ejemplo:

```
resultado = 6 % 4
puts resultado # 2
```

En el ejemplo anterior, la variable **resultado** almacena el valor 2, que es el residuo de dividir 6 entre 4.

6. **: Eleva un valor a una potencia.

Ejemplo:

```
resultado = 2 ** 3
puts resultado # 8
```

En el ejemplo anterior, la variable **resultado** almacena el valor 8, que es el resultado de elevar 2 a la potencia de 3.

4.4.2 Operadores de Comparación

Los operadores de comparación se utilizan para comparar dos valores y determinar si son iguales, diferentes, mayores, menores, etc. A continuación, se muestran algunos de los operadores de comparación más comunes en Ruby:

1. ==: Comprueba si dos valores son iguales.

Ejemplo:

```
resultado = 1 == 1
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 1 es igual a 1.

2. !=: Comprueba si dos valores son diferentes.

Ejemplo:

```
resultado = 1 != 2
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 1 es diferente de 2.

3. <: Comprueba si un valor es menor que otro.

Ejemplo:

```
resultado = 1 < 2
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 1 es menor que 2.

4. >: Comprueba si un valor es mayor que otro.

Ejemplo:

```
resultado = 2 > 1
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 2 es mayor que 1.

5. <=: Comprueba si un valor es menor o igual que otro.

Ejemplo:

```
resultado = 1 <= 1
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 1 es menor o igual que 1.

6. >=: Comprueba si un valor es mayor o igual que otro.

Ejemplo:

```
resultado = 2 >= 1
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que 2 es mayor o igual que 1.

4.4.3 Operadores Lógicos

Los operadores lógicos se utilizan para combinar expresiones lógicas y determinar si una condición es verdadera o falsa. A continuación, se muestran algunos de los operadores lógicos más comunes en Ruby:

1. **&&**: Operador AND. Devuelve true si ambas expresiones son verdaderas.

Ejemplo:

```
resultado = true && true
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que ambas expresiones son verdaderas.

2. ||: Operador OR. Devuelve true si al menos una de las expresiones es verdadera.

Ejemplo:

```
resultado = true || false
puts resultado # true
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que al menos una de las expresiones es verdadera.

3. **!**: Operador NOT. Devuelve true si la expresión es falsa y viceversa.

Ejemplo:

```
resultado = !false
puts resultado # true

resultado = !true
puts resultado # false
```

En el ejemplo anterior, la variable **resultado** almacena el valor true, que indica que la expresión es falsa.

4.4.4 Ejercicios Prácticos

1. Declara una variable de tipo Integer y asigna un valor.

Respuesta

```
numero = 5
puts numero # 5
```

En el ejemplo anterior, la variable **numero** almacena el valor 5, que es un número entero.

2. Declara una variable de tipo Float y asigna un valor.

Respuesta

```ruby
numero = 5.5
puts numero # 5.5
```

En el ejemplo anterior, la variable **numero** almacena el valor 5.5, que es un número decimal.

3. Declara una variable de tipo String y asigna un valor.

Respuesta

```ruby
nombre = "Juan"
puts nombre # Juan
```

En el ejemplo anterior, la variable **nombre** almacena el valor "Juan", que es una cadena de texto.

4. Declara una variable de tipo Symbol y asigna un valor.

Respuesta

```ruby
nombre = :juan
puts nombre # juan
```

En el ejemplo anterior, la variable **nombre** almacena el valor :juan, que es un símbolo.

5. Declara una variable de tipo Array y asigna un valor.

Respuesta

```ruby
numeros = [1, 2, 3, 4, 5]
puts numeros # [1, 2, 3, 4, 5]
```

En el ejemplo anterior, la variable **numeros** almacena un array con los valores 1, 2, 3, 4 y 5.

6. Declara una variable de tipo Hash y asigna un valor.

4.4.5 Conclusiones

En este capítulo hemos aprendido cómo se declaran variables en Ruby y los tipos de datos que podemos almacenar en ellas. También hemos visto algunos de los operadores más comunes en Ruby y cómo se utilizan para realizar operaciones aritméticas, lógicas y de comparación. Ahora que conocemos los fundamentos de las variables y los tipos de datos en Ruby, estamos listos para comenzar a trabajar con ellos en nuestros programas.

Part III

Unidad 3: Métodos y Bloques

5 Definición y Uso de Métodos

5.1 Introducción

En esta lección aprenderemos sobre la importancia de los métodos en la programación, cómo definir y utilizar métodos en Ruby, y cómo utilizar bloques, Procs, y Lambdas.

5.1.1 Definición de Métodos

Un método es un bloque de código que realiza una tarea específica. Los métodos son una parte fundamental de la programación orientada a objetos, ya que permiten reutilizar código y mantenerlo organizado.

En Ruby, los métodos se definen utilizando la palabra clave **def**, seguida del nombre del método y los parámetros que recibe. El cuerpo del método se define con un bloque de código que se ejecuta cuando se llama al método.

5.1.2 Uso de Métodos

Para llamar a un método en Ruby, simplemente escribimos su nombre seguido de paréntesis que contienen los argumentos que se le pasan al método. Por ejemplo:

```ruby
def saludar(nombre)
  puts "Hola, #{nombre}!"
end

saludar("Juan") # Hola, Juan!
```

En este ejemplo, el método **saludar** recibe un argumento **nombre** y lo imprime en la consola. Al llamar al método **saludar** con el argumento **"Juan"**, se imprimirá **"Hola, Juan!"**.

5.1.3 Valores de Retorno

Los métodos en Ruby pueden devolver un valor utilizando la palabra clave **return**. Por ejemplo:

```ruby
def suma(a, b)
  return a + b
end

resultado = suma(3, 5)
puts resultado # 8
```

En este ejemplo, el método **suma** recibe dos argumentos **a** y **b**, y devuelve la suma de los dos valores. Al llamar al método **suma** con los argumentos **3** y **5**, se asigna el resultado a la variable **resultado** y se imprime en la consola.

5.2 Bloques, Procs, y Lambdas

En Ruby, los bloques, Procs, y Lambdas son formas de encapsular código y pasarlo como argumento a un método. Aunque tienen similitudes, también tienen diferencias importantes en su comportamiento y uso.

5.2.1 Bloques

Un bloque es un fragmento de código que se puede pasar a un método y ejecutar dentro de ese método. Los bloques se definen entre llaves **{}** o con la palabra clave **do** y **end**. Por ejemplo:

```ruby
def ejecutar
  yield
end

ejecutar { puts "Hola, mundo!" } # Hola, mundo!
```

En este ejemplo, el método **ejecutar** recibe un bloque y lo ejecuta utilizando la palabra clave **yield**. Al llamar al método **ejecutar** con un bloque que imprime **"Hola, mundo!"**, se ejecutará el bloque dentro del método.

5.2.2 Procs

Un Proc es un objeto que encapsula un bloque de código y se puede almacenar en una variable para ser reutilizado. Los Procs se crean utilizando la clase **Proc** y el método **proc** o **lambda**. Por ejemplo:

```ruby
saludar = Proc.new { |nombre| puts "Hola, #{nombre}!" }
saludar.call("Juan") # Hola, Juan!
```

En este ejemplo, se crea un Proc **saludar** que recibe un argumento **nombre** y lo imprime en la consola. Al llamar al Proc **saludar** con el argumento **"Juan"**, se ejecutará el bloque del Proc.

5.2.3 Lambdas

Un Lambda es similar a un Proc, pero con algunas diferencias en su comportamiento. Los Lambdas se crean utilizando la clase **Proc** y el método **lambda**. Por ejemplo:

```
saludar = lambda { |nombre| puts "Hola, #{nombre}!" }
saludar.call("Juan") # Hola, Juan!
```

En este ejemplo, se crea un Lambda **saludar** que recibe un argumento **nombre** y lo imprime en la consola. Al llamar al Lambda **saludar** con el argumento **"Juan"**, se ejecutará el bloque del Lambda.

5.2.4 Diferencias entre Procs y Lambdas

Las diferencias más importantes entre Procs y Lambdas son:

- Los Lambdas comprueban el número de argumentos que se les pasan, mientras que los Procs no.
- Los Lambdas devuelven el valor de la última expresión evaluada, mientras que los Procs devuelven el valor de la última expresión evaluada en el método que los contiene.

Ejemplo:

```
def ejemplo
  proc { return "Proc" }.call
  lambda { return "Lambda" }.call
  return "Método"
end

puts ejemplo # Proc
```

En este ejemplo, el método **ejemplo** contiene un Proc y un Lambda que devuelven un valor. Al llamar al método **ejemplo**, se imprimirá **"Proc"** porque el Proc devuelve el valor de la última expresión evaluada en el método que lo contiene.

5.3 Creación y Uso de Métodos

Para crear un método en Ruby, utilizamos la palabra clave **def**, seguida del nombre del método y los parámetros que recibe. El cuerpo del método se define con un bloque de código que se ejecuta cuando se llama al método. Por ejemplo:

```
def saludar(nombre)
    puts "Hola, #{nombre}!"
end

saludar("Juan") # Hola, Juan!
```

En este ejemplo, el método **saludar** recibe un argumento **nombre** y lo imprime en la consola. Al llamar al método **saludar** con el argumento **"Juan"**, se imprimirá **"Hola, Juan!"**.

5.4 Métodos de Ruby

Ruby proporciona una serie de métodos integrados que se pueden utilizar para realizar tareas comunes. Algunos de los métodos más utilizados en Ruby son:

- **puts**: Imprime un mensaje en la consola.

```
puts "Hola, mundo!" # Hola, mundo!
```

En este ejemplo, el método **puts** imprime **"Hola, mundo!"** en la consola.

- **gets**: Lee una línea de texto de la consola.

```
puts "Por favor, ingresa tu nombre:"
nombre = gets.chomp
puts "Hola, #{nombre}!" # Hola, Juan!
```

En este ejemplo, el método **gets** lee una línea de texto de la consola y la asigna a la variable **nombre**.

- **chomp**: Elimina el salto de línea al final de una cadena.

```
puts "Por favor, dime tu nombre:"
nombre = gets.chomp
puts "Hola, #{nombre}!" # Hola, Juan!
```

En este ejemplo, el método **chomp** elimina el salto de línea al final de la cadena leída por el método **gets**.

- **to_i**: Convierte una cadena en un entero.

```
puts "Por favor, ingrese un número:"
numero = gets.chomp.to_i
puts numero + 1 # 6
```

En este ejemplo, el método **to_i** convierte una cadena en un número entero, que se puede utilizar en operaciones matemáticas.

- **to_f**: Convierte una cadena en un número de punto flotante.

```ruby
puts "Por favor, digite um número: "
numero = gets.chomp.to_f
puts numero + 1.5 # 6.5
```

En este ejemplo, el método **to_f** convierte una cadena en un número de punto flotante, que se puede utilizar en operaciones matemáticas con decimales.

- **to_s**: Convierte un número en una cadena.

```ruby
numero = 5
puts "El número es #{numero}" # El número es 5
puts numero.class # # Interger
numero = numero.to_s # Convertir a string
puts "El número es #{numero}" # El número es 5
puts numero.class # String
```

En este ejemplo, el método **to_s** convierte un número en una cadena, que se puede utilizar en la interpolación de cadenas.

- **length**: Devuelve la longitud de una cadena.

```ruby
cadena = "Hola, mundo!"
puts cadena.length # 12
```

En este ejemplo, el método **length** devuelve la longitud de la cadena **"Hola, mundo!"**.

- **reverse**: Invierte una cadena.

```ruby
cadena = "Hola, mundo!"
puts cadena.reverse # !odnum ,aloH
```

En este ejemplo, el método **reverse** invierte la cadena **"Hola, mundo!"**.

- **upcase**: Convierte una cadena en mayúsculas.

```ruby
cadena = "hola, mundo!"
puts cadena.upcase # HOLA, MUNDO!
```

En este ejemplo, el método **upcase** convierte la cadena **"hola, mundo!"** en mayúsculas.

- **downcase**: Convierte una cadena en minúsculas.

```ruby
cadena = "HOLA, MUNDO!"
puts cadena.downcase # hola, mundo!
```

En este ejemplo, el método **downcase** convierte la cadena **"HOLA, MUNDO!"** en minúsculas.

- **capitalize**: Convierte la primera letra de una cadena en mayúscula.

```
cadena = "hola, mundo!"
puts cadena.capitalize # Hola, mundo!
```

En este ejemplo, el método **capitalize** convierte la primera letra de la cadena **"hola, mundo!"** en mayúscula.

- **split**: Divide una cadena en un array de subcadenas.

```
cadena = "Hola, mundo!"
subcadenas = cadena.split(", ")
puts subcadenas # ["Hola", "mundo!"]
puts subcadenas[0] # Hola
puts subcadenas[1] # mundo!
```

En este ejemplo, el método **split** divide la cadena **"Hola, mundo!"** en un array de subcadenas.

- **join**: Une un array de subcadenas en una sola cadena.

```
subcadenas = ["Hola", "mundo!"]
cadena = subcadenas.join(", ")
puts cadena # Hola, mundo!
```

En este ejemplo, el método **join** une el array de subcadenas en una sola cadena.

5.5 Ejercicios Prácticos

1. Crea un método que reciba un número entero y devuelva el cuadrado del número.

Ver respuesta

```
def cuadrado(numero)
  return numero * numero
end

puts cuadrado(5) # 25
```

En este ejemplo, el método **cuadrado** recibe un número entero y devuelve el cuadrado del número.

2. Crea un método que reciba un número entero y devuelva la suma de los dígitos del número.

Ver respuesta

```ruby
def suma_digitos(numero)
  suma = 0
  while numero > 0
    suma += numero % 10
    numero /= 10
  end
  return suma
end

puts suma_digitos(123) # 6
```

En este ejemplo, el método **suma_digitos** recibe un número entero y devuelve la suma de los dígitos del número.

3. Crea un método que reciba una cadena y devuelva la cadena invertida.

Ver respuesta

```ruby
def invertir(cadena)
  return cadena.reverse
end

puts invertir("Hola, mundo!") # !odnum ,aloH
```

En este ejemplo, el método **invertir** recibe una cadena y devuelve la cadena invertida.

4. Crea un método que reciba una cadena y devuelva la cadena en mayúsculas.

Ver respuesta

```ruby
def mayusculas(cadena)
  return cadena.upcase
end

puts mayusculas("Hola, mundo!") # HOLA, MUNDO!
```

En este ejemplo, el método **mayusculas** recibe una cadena y devuelve la cadena en mayúsculas.

5. Crea un método que reciba una cadena y devuelva la cadena en minúsculas.

Ver respuesta

```ruby
def minusculas(cadena)
  return cadena.downcase
end

puts minusculas("HOLA, MUNDO!") # hola, mundo!
```

En este ejemplo, el método **minusculas** recibe una cadena y devuelve la cadena en minúsculas.

6. Crea un programa en Ruby que contenga los siguientes métodos:

    - **saludar**: Recibe un nombre y un bloque, y ejecuta el bloque pasando el nombre como argumento.
    - **sumar**: Recibe dos números y un bloque, y ejecuta el bloque pasando la suma de los dos números como argumento.
    - **restar**: Recibe dos números y un bloque, y ejecuta el bloque pasando la resta de los dos números como argumento.
    - **multiplicar**: Recibe dos números y un bloque, y ejecuta el bloque pasando el producto de los dos números como argumento.
    - **dividir**: Recibe dos números y un bloque, y ejecuta el bloque pasando la división de los dos números como argumento.

Ver respuesta

```ruby
def saludar(nombre, &bloque)
  bloque.call(nombre)
end

def sumar(a, b, &bloque)
  bloque.call(a + b)
end

def restar(a, b, &bloque)
  bloque.call(a - b)
end

def multiplicar(a, b, &bloque)
  bloque.call(a * b)
end

def dividir(a, b, &bloque)
  bloque.call(a / b)
end

saludar("Juan") { |nombre| puts "Hola, #{nombre}!" } # Hola, Juan!
sumar(3, 5) { |resultado| puts "La suma es #{resultado}" } # La suma es 8
restar(8, 3) { |resultado| puts "La resta es #{resultado}" } # La resta es 5
multiplicar(4, 6) { |resultado| puts "El producto es #{resultado}" } # El producto es 24
dividir(10, 2) { |resultado| puts "La división es #{resultado}" } # La división es 5
```

En este ejemplo, se definen los métodos **saludar**, **sumar**, **restar**, **multiplicar**, y **dividir** que reciben argumentos y un bloque, y ejecutan el bloque pasando el resultado de la operación como argumento.

7. Crea un bloque que imprima un mensaje con el nombre recibido como argumento.

Ver respuesta

```ruby
def saludar(nombre, &bloque)
  bloque.call(nombre)
end

saludar("Juan") { |nombre| puts "Hola, #{nombre}!" } # Hola, Juan!
```

En este ejemplo, se crea un bloque que imprime un mensaje con el nombre recibido como argumento.

8. Crea un bloque que imprima la suma de los dos números recibidos como argumento.

Ver respuesta

```ruby
def sumar(a, b, &bloque)
  bloque.call(a + b)
end

sumar(3, 5) { |resultado| puts "La suma es #{resultado}" } # La suma es 8
```

En este ejemplo, se crea un bloque que imprime la suma de los dos números recibidos como argumento.

9. Crea un bloque que imprima la resta de los dos números recibidos como argumento.

Ver respuesta

```ruby
def restar(a, b, &bloque)
  bloque.call(a - b)
end
restar(8, 3) { |resultado| puts "La resta es #{resultado}" } # La resta es 5
```

En este ejemplo, se crea un bloque que imprime la resta de los dos números recibidos como argumento.

10. Crea un bloque que imprima el producto de los dos números recibidos como argumento.

Ver respuesta

```ruby
def multiplicar(a, b, &bloque)
  bloque.call(a * b)
end

multiplicar(4, 6) { |resultado| puts "El producto es #{resultado}" } # El producto es 24
```

En este ejemplo, se crea un bloque que imprime el producto de los dos números recibidos como argumento.

5.6 Conclusiones

En esta lección hemos aprendido sobre la importancia de los métodos en la programación, cómo definir y utilizar métodos en Ruby, y cómo utilizar bloques, Procs, y Lambdas. Los métodos son una parte fundamental de la programación orientada a objetos, ya que permiten reutilizar código y mantenerlo organizado. Los bloques, Procs, y Lambdas son formas de encapsular código y pasarlo como argumento a un método, y tienen diferencias importantes en su comportamiento y uso. Los bloques se definen entre llaves **{}** o con la palabra clave **do** y **end**, los Procs se crean utilizando la clase **Proc** y el método **proc** o **lambda**, y los Lambdas se crean utilizando la clase **Proc** y el método **lambda**. Los Procs y Lambdas se pueden utilizar para encapsular código y pasarlo como argumento a un método, y tienen diferencias en la comprobación de argumentos y el retorno de valores. En resumen, los métodos, bloques, Procs, y Lambdas son herramientas poderosas que nos permiten escribir código más limpio, modular, y reutilizable en Ruby.

Part IV

Unidad 4: Programación Orientada a Objetos

6 Clases y Objetos

6.1 Introducción

En este capítulo se abordarán los conceptos básicos de la Programación Orientada a Objetos (POO) en Ruby. Se explicará cómo se definen las clases y cómo se crean objetos a partir de ellas. Se verán los conceptos de herencia y polimorfismo, y cómo se implementan en Ruby. También se explicará el uso de módulos y mixins para compartir comportamiento entre clases.

6.2 Conceptos básicos de la POO

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en la definición de clases y objetos. Una clase es una plantilla que define la estructura y el comportamiento de un objeto, mientras que un objeto es una instancia de una clase.

En la POO, los objetos son entidades que tienen un estado (atributos) y un comportamiento (métodos). Los atributos representan las características de un objeto, mientras que los métodos representan las acciones que un objeto puede realizar.

En Ruby, todo es un objeto. Cada valor en Ruby es un objeto, incluso los números y las cadenas de texto. Por ejemplo, en Ruby podemos hacer operaciones con números como si fueran objetos:

```
puts 1 + 2
```

En este caso, **1** y **2** son objetos de la clase **Fixnum** que representan los números **1** y **2**, respectivamente. El operador + es un método de la clase **Fixnum** que suma dos números.

6.3 Creación de Clases y Objetos

En Ruby, las clases se definen con la palabra clave **class** seguida del nombre de la clase y el cuerpo de la clase. Por ejemplo, la siguiente clase define un objeto de tipo **Vehiculo** con un atributo **marca**, **modelo** y un método **mostrar**:

```ruby
class Vehiculo
    def initialize(marca, modelo)
        @marca = marca
        @modelo = modelo
    end

    def mostrar
        puts "Marca: #{@marca}"
        puts "Modelo: #{@modelo}"
    end
end
```

En este caso, la clase **Vehiculo** tiene un método **initialize** que inicializa los atributos **marca** y **modelo** y un método **mostrar** que imprime los atributos **marca** y **modelo**.

```ruby
vehiculo = Vehiculo.new("Toyota", "Corolla")
vehiculo.mostrar # Marca: Toyota, Modelo: Corolla
```

En este caso, se crea un objeto de tipo **Vehiculo** con la marca **"Toyota"** y el modelo **"Corolla"** y se llama al método **mostrar** para imprimir los atributos **marca** y **modelo**.

6.4 Atributos y Métodos de Instancia

Los atributos de un objeto se definen con el prefijo *** seguido del nombre del atributo. Los métodos de un objeto se definen con la palabra clave** def** seguida del nombre del método y el cuerpo del método.

En el ejemplo anterior, el atributo **nombre** se define con el prefijo **(nombre?)** y el método **saludar** se define con la palabra clave **def** seguida del nombre del método y el cuerpo del método.

7 Ejemplos Prácticos

7.1 Ejemplo 1: Crear una clase y un objeto

En este ejemplo, se creará una clase **Perro** con un atributo **nombre** y un método **ladrar**. Se creará un objeto de tipo **perro** con el nombre **"Firulais"** y se llamará al método **ladrar** para imprimir el mensaje **"Guau, guay, mi nombre es Firulais"**.

```
class Perro
    def initialize(nombre)
        @nombre = nombre
    end

    def ladrar
        puts "Guau, guau, mi nombre es #{@nombre}"
    end
end

perro = Perro.new("Firulais")
perro.ladrar # Guau, guau, mi nombre es Firulais
```

En este caso, la clase **Perro** tiene un método **initialize** que inicializa el atributo **nombre** y un método **ladrar** que imprime un mensaje. Se crea un objeto de tipo **Perro** con el nombre **"Firulais"** y se llama al método **ladrar** para imprimir el mensaje **"Guau, guau, mi nombre es Firulais"**.

7.2 Ejemplo 2: Crear una clase con herencia

En este ejemplo, se creará una clase **Deportes** con un atributo **nombre** y un método **practicar**. Se creará una clase **Futbol** que hereda de la clase **Deportes** y tiene un atributo **equipo** y un método **jugar**. Se creará un objeto de tipo **Futbol** con el nombre **"Lionel Messi"** y el equipo **"Barcelona"** y se llamará al método **practicar** para imprimir el mensaje **"Estoy practicando Lionel Messi"**.

```
class Deportes
    def initialize(nombre)
        @nombre = nombre
    end

    def practicar
```

```
        puts "Estoy practicando #{@nombre}"
    end
end

class Futbol < Deportes
    def initialize(nombre, equipo)
        super(nombre)
        @equipo = equipo
    end

    def jugar
        puts "Estoy jugando al futbol en el equipo #{@equipo}"
    end
end

futbolista = Futbol.new("Lionel Messi", "Barcelona")
futbolista.practicar # Estoy practicando Lionel Messi
```

En este caso, la clase **Deportes** tiene un método **initialize** que inicializa el atributo **nombre** y un método **practicar** que imprime un mensaje. La clase **Futbol** hereda de la clase **Deportes** y tiene un método **initialize** que inicializa el atributo **equipo** y un método **jugar** que imprime un mensaje.

7.3 Ejemplo 3: Crear un módulo y compartir comportamiento

En este ejemplo, se creará un módulo **Saludar** con un método **saludar** que imprime un mensaje. El módulo **Saludar** se incluirá en las clases **Persona** y **Empleado** para compartir el comportamiento. Se crearán objetos de tipo **Persona** y **Empleado** y se llamará al método **saludar** para imprimir el mensaje **"Hola, soy una persona"**.

```
class Persona
    def saludar
        puts "Hola, soy una persona"
    end
end

class Empleado
    def saludar
        puts "Hola, soy un empleado"
    end
end

module Saludar
    def saludar
        puts "Hola, soy una persona"
    end
```

```ruby
end

class Persona
    include Saludar
end

class Empleado
    include Saludar
end

persona = Persona.new
empleado = Empleado.new

persona.saludar # Hola, soy una persona
empleado.saludar # Hola, soy una persona
```

En este caso, el módulo **Saludar** tiene un método **saludar** que imprime un mensaje. El módulo **Saludar** se incluye en las clases **Persona** y **Empleado** con la palabra clave **include** para compartir el comportamiento. Se crean objetos de tipo **Persona** y **Empleado** y se llama al método **saludar** para imprimir el mensaje **"Hola, soy una persona"**.

8 Ejercicios Prácticos

1. Crear una clase **Rectángulo** con atributos **base** y **altura** y métodos para calcular el área y el perímetro del rectángulo.
2. Crear una clase **Cuadrado** que hereda de la clase **Rectángulo** y tiene un método para calcular el área y el perímetro del cuadrado.
3. Crear una clase **Círculo** con atributo **radio** y métodos para calcular el área y la circunferencia del círculo.
4. Crear una clase **Triángulo** con atributos **base** y **altura** y métodos para calcular el área y el perímetro del triángulo.
5. Crear una clase **Figura** con métodos para calcular el área y el perímetro de una figura geométrica genérica.

8.0.1 Herencia y Polimorfismo

En Ruby, la herencia y el polimorfismo son conceptos fundamentales de la Programación Orientada a Objetos (POO). La herencia permite que una clase herede los atributos y métodos de otra clase, mientras que el polimorfismo permite que un objeto se comporte de diferentes maneras según el contexto.

En Ruby, la herencia se define con la palabra clave $<$ seguida del nombre de la clase de la que se hereda. Por ejemplo, la siguiente clase **Empleado** hereda de la clase **Persona**:

```ruby
class Profesion
    def initialize(profesion)
        @profesion = profesion
    end

    def mostrar
        puts "Profesión: #{@profesion}"
    end
end

class Persona < Profesion
    def initialize(nombre, profesion)
        super(profesion)
        @nombre = nombre
    end

    def saludar
        puts "Hola, mi nombre es #{@nombre}"
    end
```

```ruby
end

persona = Persona.new("Juan", "Ingeniero")
persona.saludar # Hola, mi nombre es Juan
persona.mostrar # Profesión: Ingeniero
```

En este caso, la clase **Persona** hereda de la clase **Profesion** y tiene un método **initialize** que inicializa los atributos **nombre** y **profesion** y un método **saludar** que imprime un mensaje. Se crea un objeto de tipo **Persona** con el nombre **"Juan"** y la profesión **"Ingeniero"** y se llama a los métodos **saludar** y **mostrar** para imprimir los mensajes.

8.0.2 Polimorfismo

El polimorfismo es la capacidad de un objeto de comportarse de diferentes maneras según el contexto. En Ruby, el polimorfismo se logra mediante el uso de métodos polimórficos que se comportan de manera diferente según el tipo de objeto.

En el siguiente ejemplo, se define un método **hablar** en la clase **Animal** que lanza una excepción si se llama directamente. Las clases **Perro** y **Gato** heredan de la clase **Animal** y tienen un método **hablar** que imprime un mensaje. Se crean objetos de tipo **Perro** y **Gato** y se llama al método **hablar** para imprimir los mensajes **"Guau, guau"** y **"Miau, miau"**:

```ruby
class Animal
    def hablar
        raise NotImplementedError, "Subclass must implement abstract method"
    end
end

class Perro < Animal
    def hablar
        puts "Guau, guau"
    end
end

class Gato < Animal
    def hablar
        puts "Miau, miau"
    end
end

perro = Perro.new
gato = Gato.new

perro.hablar # Guau, guau
gato.hablar # Miau, miau
```

En este caso, la clase **Animal** tiene un método **hablar** que lanza una excepción si se llama directamente. Las clases **Perro** y **Gato** heredan de la clase **Animal** y tienen un método **hablar** que imprime un mensaje. Se crean objetos de tipo **Perro** y **Gato** y se llama al método **hablar** para imprimir los mensajes **"Guau, guau"** y **"Miau, miau"**.

8.0.3 Módulos y Mixins

En Ruby, los módulos son una forma de compartir comportamiento entre clases. Un módulo es un contenedor de métodos y constantes que se pueden incluir en una clase para agregar funcionalidad. Los módulos se incluyen en una clase con la palabra clave **include**.

En el siguiente ejemplo, se define un módulo **Saludar** con un método **saludar** que imprime un mensaje. El módulo **Saludar** se incluye en las clases **Persona** y **Empleado** para compartir el comportamiento:

```
module Saludar
    def saludar
        puts "Hola, soy una persona"
    end
end

class Persona
    include Saludar
end

class Empleado
    include Saludar
end

persona = Persona.new
empleado = Empleado.new

persona.saludar # Hola, soy una persona
empleado.saludar # Hola, soy una persona
```

En este caso, el módulo **Saludar** tiene un método **saludar** que imprime un mensaje. El módulo **Saludar** se incluye en las clases **Persona** y **Empleado** con la palabra clave **include** para compartir el comportamiento. Se crean objetos de tipo **Persona** y **Empleado** y se llama al método **saludar** para imprimir el mensaje **"Hola, soy una persona"**.

8.0.4 Ejercicios Prácticos

1. Crear una clase **Vehículo** con atributos **marca** y **modelo** y métodos para mostrar la marca y el modelo del vehículo.
2. Crear una clase **Automóvil** que hereda de la clase **Vehículo** y tiene un atributo **color** y un método para mostrar el color del automóvil.

3. Crear una clase **Motocicleta** que hereda de la clase **Vehículo** y tiene un atributo **cilindrada** y un método para mostrar la cilindrada de la motocicleta.
4. Crear una clase **Camión** que hereda de la clase **Vehículo** y tiene un atributo **carga** y un método para mostrar la carga del camión.
5. Crear una clase **Transporte** con métodos para mostrar la marca y el modelo de un vehículo genérico.

9 Conclusiones

En este capítulo se han abordado los conceptos básicos de la Programación Orientada a Objetos (POO) en Ruby. Se ha explicado cómo se definen las clases y cómo se crean objetos a partir de ellas. Se han visto los conceptos de herencia y polimorfismo, y cómo se implementan en Ruby. También se ha explicado el uso de módulos y mixins para compartir comportamiento entre clases.

En resumen, la POO es un paradigma de programación que se basa en la definición de clases y objetos. En Ruby, las clases se definen con la palabra clave **class** seguida del nombre de la clase y el cuerpo de la clase. Los objetos se crean a partir de una clase con el método **new** y se pueden llamar a los métodos de la clase con el operador **.**. La herencia permite que una clase herede los atributos y métodos de otra clase, mientras que el polimorfismo permite que un objeto se comporte de diferentes maneras según el contexto. Los módulos y mixins permiten compartir comportamiento entre clases.

En los ejemplos prácticos se han creado clases y objetos para representar diferentes entidades y se ha mostrado cómo se pueden compartir comportamientos entre clases. Se han creado clases con herencia y polimorfismo para mostrar cómo se pueden reutilizar y extender funcionalidades. Se han creado módulos y mixins para compartir comportamientos entre clases y se ha mostrado cómo se pueden incluir en las clases para agregar funcionalidades.

Part V

Unidad 5: Manejo de Errores y Pruebas

10 Manejo de Excepciones

10.1 Introducción

En esta sección aprenderemos a manejar errores y excepciones en Ruby. Veremos cómo podemos rescatar excepciones y cómo podemos crear nuestras propias excepciones personalizadas. También veremos cómo podemos realizar pruebas unitarias en Ruby y cómo podemos utilizar herramientas de debugging para encontrar errores en nuestro código.

10.2 Rescate de Excepciones

En Ruby, las excepciones son objetos que representan un error en la ejecución de un programa. Cuando se produce un error, Ruby lanza una excepción y detiene la ejecución del programa. Para evitar que el programa se detenga, podemos rescatar excepciones utilizando la palabra clave **rescue**.

```
begin
    # Código que puede lanzar una excepción
    file = File.open("data.txt")
    contents = file.read
    puts contents
rescue Errno::ENOENT
    # Código que se ejecuta si el archivo no existe
    puts "El archivo no existe"
rescue Errno::EACCES
    # Código que se ejecuta si no se tiene acceso al archivo
    puts "No se tiene acceso al archivo"
rescue => e

    # Código que se ejecuta para cualquier otra excepción
    puts "Se ha producido un error: #{e.message}"
ensure

    # Código que se ejecuta siempre, sin importar si se lanzó una excepción o no
    file.close if file
end
```

En el ejemplo anterior, el bloque **begin** se utiliza para envolver el código que puede lanzar una excepción. Si se produce una excepción, el bloque **rescue** se ejecuta y se captura la excepción. Podemos utilizar varios bloques **rescue** para capturar diferentes

excepciones. También podemos utilizar un bloque **rescue** sin argumentos para capturar cualquier excepción. El bloque **ensure** se ejecuta siempre, sin importar si se lanzó una excepción o no.

10.3 Creación de Excepciones Personalizadas

En Ruby, podemos crear nuestras propias excepciones personalizadas heredando de la clase **Exception**.

```
class MyError < Exception
end

begin
    raise MyError, "Se ha producido un error"
rescue MyError => e
    puts "Se ha producido un error personalizado: #{e.message}"
end
```

En el ejemplo anterior, creamos una excepción personalizada llamada **MyError** que hereda de la clase **Exception**. Luego, lanzamos la excepción **MyError** utilizando la palabra clave **raise** y capturamos la excepción utilizando un bloque **rescue**.

Ejemplo práctico: Crear una excepción personalizada llamada **NegativeNumberError** que se lanza cuando se intenta calcular la raíz cuadrada de un número negativo.

```
class NegativeNumberError < Exception
end

def square_root(x)
    raise NegativeNumberError, "No se puede calcular la raíz cuadrada de un número negati
    Math.sqrt(x)
end

begin
    puts square_root(-1)
rescue NegativeNumberError => e
    puts "Se ha producido un error: #{e.message}"
end
```

10.3.0.1 Pruebas y Debugging

En Ruby, podemos realizar pruebas unitarias utilizando la biblioteca **MiniTest** o **RSpec**. También podemos utilizar las gemas **pry** y **byebug** para realizar debugging en nuestro código.

```ruby
require 'minitest/autorun'

class NegativeNumberError < StandardError; end

def square_root(number)
  raise NegativeNumberError if number.negative?

  Math.sqrt(number)
end

class TestSquareRoot < Minitest::Test
  def test_square_root
    assert_equal 2, square_root(4)
    assert_equal 0, square_root(0)
    assert_raises NegativeNumberError do
      square_root(-1)
    end
  end
end
```

En el ejemplo anterior, creamos una prueba unitaria utilizando la biblioteca **MiniTest** para la función **square_root**. La prueba verifica que se lanza una excepción **NegativeNumberError** cuando se intenta calcular la raíz cuadrada de un número negativo.

Ejemplo práctico: Crear una prueba unitaria para la función **square_root** que verifica que se lanza una excepción **NegativeNumberError** cuando se intenta calcular la raíz cuadrada de un número negativo.

```ruby
require 'minitest/autorun'

class NegativeNumberError < StandardError
end

def square_root(number)
  raise NegativeNumberError if number < 0

  Math.sqrt(number)
end

class TestSquareRoot < Minitest::Test
  def test_square_root
    assert_equal 2, square_root(4)
    assert_equal 0, square_root(0)
    assert_raises NegativeNumberError do
      square_root(-1)
    end
  end
end
```

En el ejemplo anterior, creamos una prueba unitaria utilizando la biblioteca **MiniTest** para la función **square_root**. La prueba verifica que se lanza una excepción **NegativeNumberError** cuando se intenta calcular la raíz cuadrada de un número negativo.

10.3.0.2 Ejercicios Prácticos

1. Crear una excepción personalizada llamada **InvalidEmailError** que se lanza cuando se intenta crear un objeto **Email** con una dirección de correo electrónico inválida.

Ver respuesta

```
class InvalidEmailError < Exception
end
```

En el ejercicio anterior, creamos una excepción personalizada llamada **InvalidEmailError** que hereda de la clase **Exception**.

2. Crear una clase **Email** que tiene un atributo **address** que representa la dirección de correo electrónico. La clase debe tener un constructor que recibe la dirección de correo electrónico y lanza una excepción **InvalidEmailError** si la dirección de correo electrónico es inválida.

Ver respuesta

```
class Email
  attr_reader :address

  def initialize(address)
    raise InvalidEmailError, "Dirección de correo electrónico inválida" unless address.ma

    @address = address
  end
end

begin
  email = Email.new("
    puts email.address
rescue InvalidEmailError => e
    puts "Se ha producido un error: #{e.message}"
    end
```

En el ejercicio anterior, creamos una clase **Email** que tiene un atributo **address** que representa la dirección de correo electrónico. El constructor de la clase lanza una excepción **InvalidEmailError** si la dirección de correo electrónico es inválida.

3. Crear una prueba unitaria para la clase **Email** que verifica que se lanza una excepción **InvalidEmailError** cuando se intenta crear un objeto **Email** con una dirección de correo electrónico inválida.

Ver respuesta

```
require 'minitest/autorun'

class InvalidEmailError < StandardError; end

class Email
  attr_reader :address

  def initialize(address)
    raise InvalidEmailError, "Dirección de correo electrónico inválida" unless address.ma

    @address = address
  end
end

class TestEmail < Minitest::Test
  def test_email
    assert_raises InvalidEmailError do
      Email.new("invalid_email")
    end
  end
end
```

En el ejercicio anterior, creamos una prueba unitaria utilizando la biblioteca **MiniTest** para la clase **Email**. La prueba verifica que se lanza una excepción **InvalidEmailError** cuando se intenta crear un objeto **Email** con una dirección de correo electrónico inválida.

4. Crear una excepción personalizada llamada **InvalidPasswordError** que se lanza cuando se intenta crear un objeto **Password** con una contraseña inválida.

Ver respuesta

```
class InvalidPasswordError < Exception
end
```

En el ejercicio anterior, creamos una excepción personalizada llamada **InvalidPasswordError** que hereda de la clase **Exception**.

5. Crear una clase **Password** que tiene un atributo **value** que representa la contraseña. La clase debe tener un constructor que recibe la contraseña y lanza una excepción **InvalidPasswordError** si la contraseña es inválida.

Ver respuesta

```
class Password
  attr_reader :value

  def initialize(value)
```

```
    raise InvalidPasswordError, "Contraseña inválida" unless value.match?(/\A(?=.*[a-z])(

    @value = value
  end
end

begin
  password = Password.new("password")
  puts password.value
rescue InvalidPasswordError => e
    puts "Se ha producido un error: #{e.message}"
    end
```

En el ejercicio anterior, creamos una clase **Password** que tiene un atributo **value** que representa la contraseña. El constructor de la clase lanza una excepción **InvalidPasswordError** si la contraseña es inválida.

11 Conclusiones

En esta sección aprendimos a manejar errores y excepciones en Ruby. Vimos cómo podemos rescatar excepciones y cómo podemos crear nuestras propias excepciones personalizadas. También aprendimos a realizar pruebas unitarias en Ruby y a utilizar herramientas de debugging para encontrar errores en nuestro código.

Part VI

Unidad 6: Ruby Avanzado

12 Metaprogramación y Enumerables en Ruby

12.1 Introducción

Ruby es un lenguaje de programación que permite la metaprogramación, es decir, la capacidad de un programa de modificar su estructura y comportamiento en tiempo de ejecución. Esto se logra a través de la reflexión, que es la capacidad de un programa de examinar y modificar su propia estructura y comportamiento.

En esta sección se abordarán los conceptos de metaprogramación y enumerables en Ruby, así como su aplicación en la resolución de problemas.

12.2 Metaprogramación

La metaprogramación es una técnica de programación que permite a un programa modificar su estructura y comportamiento en tiempo de ejecución. En Ruby, la metaprogramación se logra a través de la reflexión, que es la capacidad de un programa de examinar y modificar su propia estructura y comportamiento.

En Ruby, la metaprogramación se logra a través de la reflexión, que es la capacidad de un programa de examinar y modificar su propia estructura y comportamiento. Algunas de las técnicas de metaprogramación en Ruby son:

- Definición dinámica de métodos: Permite agregar métodos a una clase en tiempo de ejecución.

Ejemplo:

```ruby
class MyClass
  define_method :my_method do
    puts "Hello, world!"
  end
end

obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **my_method** de manera dinámica en la clase **MyClass** y se invoca en una instancia de la clase.

- Uso de method_missing: Permite capturar llamadas a métodos que no existen y responder a ellas de manera dinámica.

Ejemplo:

```
class MyClass
  def method_missing(name, *args)
    puts "Method #{name} not found!"
  end
end

obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **method_missing** en la clase **MyClass** que captura llamadas a métodos que no existen y responde a ellas de manera dinámica.

- Uso de send: Permite invocar métodos de manera dinámica.

Ejemplo:

```
class MyClass
  def my_method
    puts "Hello, world!"
  end
end

obj = MyClass.new
obj.send(:my_method)
```

En este ejemplo, se invoca el método **my_method** de manera dinámica en una instancia de la clase **MyClass** utilizando el método **send**.

- Uso de define_method: Permite definir métodos de manera dinámica.

Ejemplo:

```
class MyClass
  define_method :my_method do
    puts "Hello, world!"
  end
end

obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **my_method** de manera dinámica en la clase **MyClass** y se invoca en una instancia de la clase.

12.3 Enumerables y Enumeradores

Los enumerables y enumeradores son una parte fundamental de Ruby. Los enumerables son módulos que proporcionan métodos para recorrer y manipular colecciones de objetos. Los enumeradores son objetos que encapsulan la lógica de recorrido de una colección.

Algunos de los métodos comunes de Enumerables en Ruby son:

- each: Permite recorrer una colección de objetos.

Ejemplo:

```ruby
[1, 2, 3].each { |x| puts x }

(1..3).each { |x| puts x }

{ a: 1, b: 2, c: 3 }.each { |k, v| puts "#{k}: #{v}" }
```

En este ejemplo, se recorren una matriz, un rango y un hash utilizando el método **each**.

- map: Permite transformar una colección de objetos.

Ejemplo:

```ruby
[1, 2, 3].map { |x| x * 2 }

(1..3).map { |x| x * 2 }

{ a: 1, b: 2, c: 3 }.map { |k, v| [k, v * 2] }
```

En este ejemplo, se transforman una matriz, un rango y un hash utilizando el método **map**.

- select: Permite filtrar una colección de objetos.

Ejemplo:

```ruby
[1, 2, 3].select { |x| x.even? }

(1..3).select { |x| x.even? }

{ a: 1, b: 2, c: 3 }.select { |k, v| v.even? }
```

En este ejemplo, se filtran una matriz, un rango y un hash utilizando el método **select**.

- reduce: Permite combinar una colección de objetos en un único valor.

Ejemplo:

```ruby
[1, 2, 3].reduce(0) { |acc, x| acc + x }

(1..3).reduce(0) { |acc, x| acc + x }

{ a: 1, b: 2, c: 3 }.reduce({}) { |acc, (k, v)| acc.merge(k => v * 2) }
```

En este ejemplo, se combinan una matriz, un rango y un hash en un único valor utilizando el método **reduce**.

12.4 Ejercicios Prácticos

A continuación se presentan algunos ejercicios prácticos que permiten aplicar los conceptos de metaprogramación y enumerables en Ruby:

1. Implementar un método que permita definir métodos de manera dinámica en una clase.

Ver solución

```ruby
class MyClass
  def self.define_method(name, &block)
    define_method(name, &block)
  end
end

MyClass.define_method(:my_method) do
  puts "Hello, world!"
end

obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **define_method** en la clase **MyClass** que permite definir métodos de manera dinámica. Luego, se define un método llamado **my_method** de manera dinámica en la clase **MyClass** y se invoca en una instancia de la clase.

2. Implementar un método que permita capturar llamadas a métodos que no existen y responder a ellas de manera dinámica.

Ver solución

```ruby
class MyClass
  def self.method_missing(name, *args)
    puts "Method #{name} not found!"
  end
end
```

```ruby
obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **method_missing** en la clase **MyClass** que captura llamadas a métodos que no existen y responde a ellas de manera dinámica.

3. Implementar un método que permita invocar métodos de manera dinámica.

```ruby
class MyClass
  def my_method
    puts "Hello, world!"
  end
end

obj = MyClass.new
obj.send(:my_method)
```

En este ejemplo, se invoca el método **my_method** de manera dinámica en una instancia de la clase **MyClass** utilizando el método **send**.

4. Implementar un método que permita definir métodos de manera dinámica.

Ver solución

```ruby
class MyClass
  def self.define_method(name, &block)
    define_method(name, &block)
  end
end

MyClass.define_method(:my_method) do
  puts "Hello, world!"
end

obj = MyClass.new
obj.my_method
```

En este ejemplo, se define un método llamado **define_method** en la clase **MyClass** que permite definir métodos de manera dinámica. Luego, se define un método llamado **my_method** de manera dinámica en la clase **MyClass** y se invoca en una instancia de la clase.

5. Implementar un método que permita recorrer una colección de objetos y aplicar una función a cada uno de ellos.

Ver solución

```ruby
def my_each(collection, &block)
  collection.each(&block)
end

my_each([1, 2, 3]) { |x| puts x }

my_each(1..3) { |x| puts x }

my_each({ a: 1, b: 2, c: 3 }) { |k, v| puts "#{k}: #{v}" }
```

En este ejemplo, se implementa un método llamado **my_each** que permite recorrer una colección de objetos y aplicar una función a cada uno de ellos.

6. Implementar un método que permita filtrar una colección de objetos según un criterio dado.

Ver solución

```ruby
def my_select(collection, &block)
  collection.select(&block)
end

my_select([1, 2, 3]) { |x| x.even? }

my_select(1..3) { |x| x.even? }

my_select({ a: 1, b: 2, c: 3 }) { |k, v| v.even? }
```

En este ejemplo, se implementa un método llamado **my_select** que permite filtrar una colección de objetos según un criterio dado.

7. Implementar un método que permita combinar una colección de objetos en un único valor.

Ver solución

```ruby
def my_reduce(collection, initial, &block)
    collection.reduce(initial, &block)
end

my_reduce([1, 2, 3], 0) { |acc, x| acc + x }

my_reduce(1..3, 0) { |acc, x| acc + x }

my_reduce({ a: 1, b: 2, c: 3 }, {}) { |acc, (k, v)| acc.merge(k => v * 2) }
```

En este ejemplo, se implementa un método llamado **my_reduce** que permite combinar una colección de objetos en un único valor.

12.5 Conclusiones

La metaprogramación y los enumerables son técnicas fundamentales en Ruby que permiten a los programadores modificar la estructura y comportamiento de un programa en tiempo de ejecución, así como recorrer y manipular colecciones de objetos de manera eficiente. La aplicación de estos conceptos en la resolución de problemas permite escribir código más flexible, conciso y expresivo. Por lo tanto, es importante comprender y dominar estos conceptos para aprovechar al máximo las capacidades de Ruby como lenguaje de programación.

Part VII

Unidad 7: Ruby y la Web

13 Introducción a Ruby on Rails

Figure 13.1: Ruby on Rails

En esta sección se abordará una introducción a Ruby on Rails, un framework de desarrollo web que permite crear aplicaciones web de manera rápida y sencilla.

13.1 ¿Qué es Ruby on Rails?

Ruby on Rails es un framework de desarrollo web escrito en el lenguaje de programación Ruby. Fue creado por David Heinemeier Hansson y lanzado en 2004. Rails es un framework de código abierto que sigue el patrón de diseño Modelo-Vista-Controlador (MVC) y se basa en la filosofía de la convención sobre configuración.

Rails es conocido por su simplicidad y facilidad de uso, lo que lo convierte en una excelente opción para desarrollar aplicaciones web de manera rápida y eficiente. Rails proporciona una serie de herramientas y bibliotecas que facilitan el desarrollo de aplicaciones web, como la generación automática de código, la integración con bases de datos y la gestión de sesiones y cookies.

13.2 Instalación de Ruby on Rails

Para instalar Ruby on Rails en tu sistema, primero necesitas tener Ruby instalado. Puedes instalar Ruby siguiendo las instrucciones en el sitio web oficial de Ruby: https://www.ruby-lang.org/en/documentation/installation/

Una vez que tengas Ruby instalado, puedes instalar Rails utilizando el siguiente comando en tu terminal:

```
gem install rails
```

Este comando instalará la última versión de Rails en tu sistema. Una vez que la instalación haya finalizado, puedes verificar que Rails se haya instalado correctamente ejecutando el siguiente comando:

```
rails --version
```

Este comando debería mostrar la versión de Rails que has instalado en tu sistema.

14 Creación de una nueva aplicación Rails

14.1 Parte 1: Configuración del Proyecto

14.1.1 Creación del Proyecto:

Crea un nuevo proyecto de Rails llamado inventory_app:

```
rails new inventory_app
```

Esto generará la estructura básica de un proyecto de Rails.

14.1.2 Navega a la Carpeta del Proyecto:

Entra en el directorio del proyecto recién creado:

```
cd inventory_app
```

14.1.3 Inicia el Servidor:

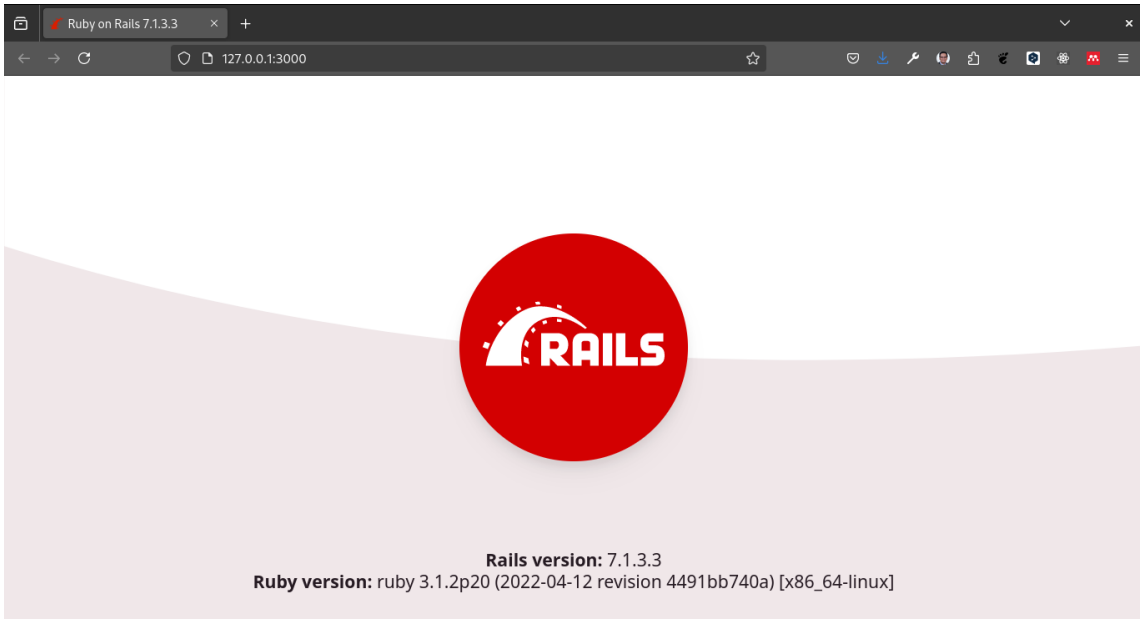


Figure 14.1: Servidor de Rails corriendo

Para asegurarte de que todo está funcionando correctamente, inicia el servidor:

```
rails server
```

Luego, abre tu navegador y ve a http://localhost:3000. Deberías ver la página de bienvenida de Rails.

14.2 Parte 2: Generación del Modelo

14.2.1 Conceptos Básicos

En Rails, un modelo representa una tabla en la base de datos. Vamos a crear un modelo llamado Item para manejar nuestro inventario.

14.2.2 Generación del Modelo:

Genera el modelo Item con atributos name, description, y quantity:

```
rails generate model Item name:string description:text quantity:integer
```

Este comando creará un archivo de migración para la tabla items y el modelo Item.

14.2.3 Ejecutar la Migración:

Aplica la migración para crear la tabla items en la base de datos:

```
rails db:migrate
```

14.3 Parte 3: Generación del Controlador

14.3.1 Conceptos Básicos

Un controlador en Rails recibe las solicitudes del usuario, interactúa con el modelo y la base de datos, y luego renderiza una vista.

14.3.2 Generación del Controlador:

Genera el controlador Items con las acciones necesarias (index, show, new, create, edit, update, destroy):

```
rails generate controller Items index show new edit
```

Este comando generará un controlador con las acciones especificadas y las vistas correspondientes.

14.4 Parte 4: Rutas

14.4.1 Conceptos Básicos

Las rutas en Rails configuran las URL de la aplicación y especifican qué controlador y acción deben manejar una solicitud determinada.

14.4.2 Configurar las Rutas:

Abre el archivo **config/routes.rb** y configura las rutas para los items:

```
Rails.application.routes.draw do
      resources :items
      root 'items#index'
    end
```

14.5 Parte 5: Vistas

14.5.1 Conceptos Básicos

Las vistas en Rails son plantillas que muestran la información al usuario. Utilizan HTML con código Ruby embebido (ERB).

14.5.2 Crear las Vistas:

Abre el archivo **app/views/items/index.html.erb** y agrega el siguiente código:

```
<h1>Inventory</h1>

<%= link_to 'New Item', new_item_path %>

<ul>
  <% @items.each do |item| %>
    <li>
      <%= link_to item.name, item_path(item) %>
      <%= link_to 'Edit', edit_item_path(item) %>
      <%= link_to 'Destroy', item, method: :delete, data: { confirm: 'Are you sure?' } %>
    </li>
  <% end %>
</ul>
```

14.5.3 Controlador y Acción index:

Abre el archivo **app/controllers/items_controller.rb** y agrega la acción index:

```
class ItemsController < ApplicationController
    def index
        @items = Item.all
    end
end
```

14.6 Parte 6: CRUD - Create

14.6.1 Conceptos Básicos

Para crear un nuevo ítem, necesitamos una vista para el formulario y una acción en el controlador para manejar la creación.

14.6.2 Formulario para Crear un Nuevo Ítem:

Abre el archivo **app/views/items/new.html.erb** y agrega el siguiente código:

```
<h1>New Item</h1>

<%= form_with model: @item, local: true do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>
  <div>
    <%= form.label :description %>
    <%= form.text_area :description %>
  </div>
  <div>
    <%= form.label :quantity %>
    <%= form.number_field :quantity %>
  </div>
  <div>
    <%= form.submit %>
  </div>
<% end %>
```

14.6.3 Acción new y create en el Controlador:

Abre **app/controllers/items_controller.rb** y agrega las acciones new y create:

```
class ItemsController < ApplicationController
    def index
    @items = Item.all
    end

    def show
    @item = Item.find(params[:id])
    end

    def new
    @item = Item.new
    end

    def create
    @item = Item.new(item_params)
    if @item.save
        redirect_to @item
    else
        render :new
    end
    end

    private

    def item_params
    params.require(:item).permit(:name, :description, :quantity)
    end
end
```

14.7 Parte 7: CRUD - Read

14.7.1 Conceptos Básicos

Para leer los detalles de un ítem, necesitamos una vista show.

14.7.2 Vista show:

Abre **app/views/items/show.html.erb** y agrega el siguiente código:

```
<h1><%= @item.name %></h1>
<p><%= @item.description %></p>
<p>Quantity: <%= @item.quantity %></p>

<%= link_to 'Edit', edit_item_path(@item) %> |
<%= link_to 'Back', items_path %>
```

14.8 Parte 8: CRUD - Update

14.8.1 Conceptos Básicos

Para actualizar un ítem, necesitamos una vista para el formulario de edición y una acción en el controlador para manejar la actualización.

14.8.2 Formulario para Editar un Ítem:

Abre **app/views/items/edit.html.erb** y agrega el siguiente código:

```
<h1>Edit Item</h1>

<%= form_with model: @item, local: true do |form| %>
<div>
<%= form.label :name %>
<%= form.text_field :name %>
</div>
<div>
<%= form.label :description %>
<%= form.text_area :description %>
</div>
<div>
<%= form.label :quantity %>
<%= form.number_field :quantity %>
</div>
<div>
<%= form.submit %>
</div>
<% end %>
```

14.8.3 Acción edit y update en el Controlador:

Abre **app/controllers/items_controller.rb** y agrega las acciones edit y update:

```
class ItemsController < ApplicationController
    def index
    @items = Item.all
    end

    def show
    @item = Item.find(params[:id])
    end

    def new
    @item = Item.new
```

```
    end

    def create
    @item = Item.new(item_params)
    if @item.save
        redirect_to @item
    else
        render :new
    end
    end

    def edit
    @item = Item.find(params[:id])
    end

    def update
    @item = Item.find(params[:id])
    if @item.update(item_params)
        redirect_to @item
    else
        render :edit
    end
    end

    private

    def item_params
    params.require(:item).permit(:name, :description, :quantity)
    end
end
```

14.8.4 Parte 9: CRUD - Delete

14.8.5 Conceptos Básicos

Para eliminar un ítem, necesitamos una acción en el controlador que maneje la eliminación.

14.8.6 Acción destroy en el Controlador:

Abre **app/controllers/items_controller.rb** y agrega la acción destroy:

```
class ItemsController < ApplicationController
    def index
    @items = Item.all
    end
```

```
    def show
    @item = Item.find(params[:id])
    end

    def new
    @item = Item.new
    end

    def create
    @item = Item.new(item_params)
    if @item.save
        redirect_to @item
    else
        render :new
    end
    end

    def edit
    @item = Item.find(params[:id])
    end

    def update
    @item = Item.find(params[:id])
    if @item.update(item_params)
        redirect_to @item
    else
        render :edit
    end
    end

    def destroy
    @item = Item.find(params[:id])
    @item.destroy
    redirect_to items_path
    end

    private

    def item_params
    params.require(:item).permit(:name, :description, :quantity)
    end
end
```

14.9 Parte 10: Pruebas y Verificación

14.9.1 Conceptos Básicos

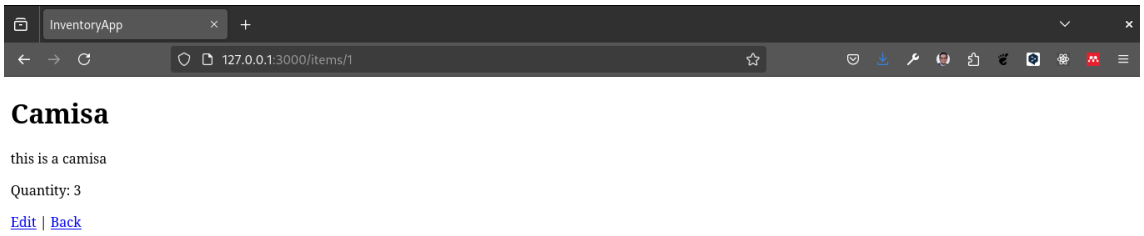
Es importante probar que nuestra aplicación funciona correctamente y que todas las operaciones CRUD están funcionando como se espera.

14.9.2 Verificar la Funcionalidad:

Inicia el servidor de Rails nuevamente (si no está ya en ejecución):

```
rails server
```

14.10 Pruebas Manuales:



Abre tu navegador y navega a http://localhost:3000/items. Deberías ver la lista de ítems.

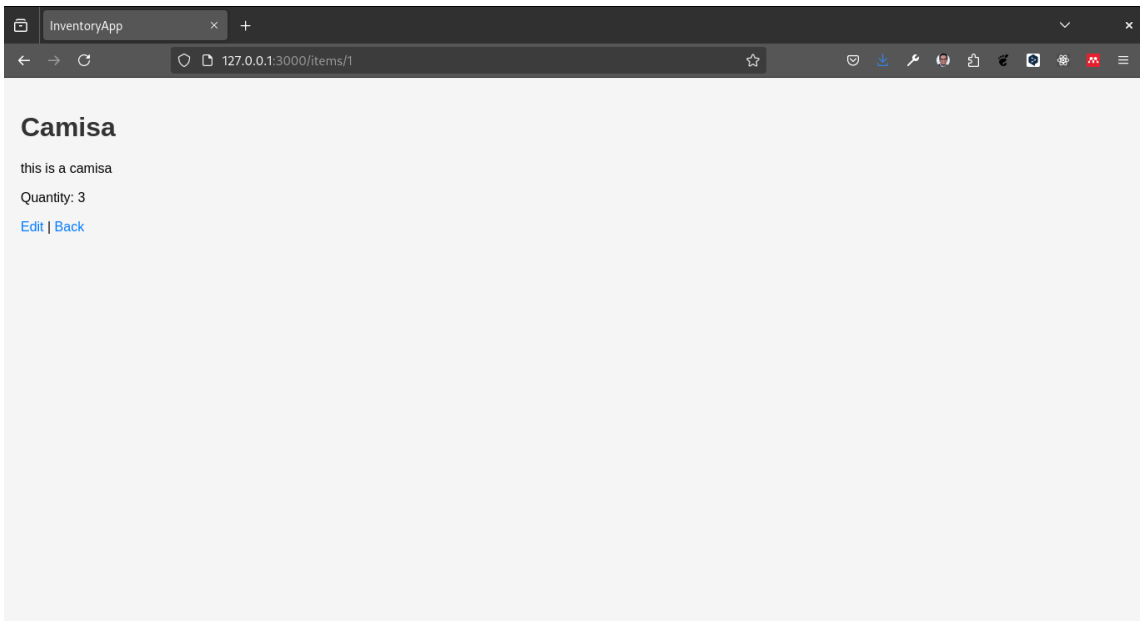
Crear un Nuevo Ítem: Haz clic en "New Item" y completa el formulario para crear un nuevo ítem.

Ver un Ítem: Haz clic en el nombre de un ítem para ver sus detalles.

Editar un Ítem: Haz clic en "Edit" junto a un ítem para editarlo.

Eliminar un Ítem: Haz clic en "Destroy" junto a un ítem para eliminarlo.

14.11 Parte 11: Estilo y Mejoras



14.11.1 Conceptos Básicos

Podemos mejorar la apariencia y la usabilidad de nuestra aplicación añadiendo algo de estilo con CSS.

14.11.2 Agregar Estilo:

Abre el archivo **app/assets/stylesheets/application.css** y agrega algunas reglas CSS básicas para mejorar el diseño:

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
    background-color: #f5f5f5;
}

h1 {
    color: #333;
}

ul {
    list-style: none;
    padding: 0;
}
```

```
li {
    padding: 10px;
    background: #fff;
    margin-bottom: 10px;
    border-radius: 5px;
    box-shadow: 0 1px 3px rgba(0,0,0,0.1);
}

a {
    color: #007bff;
    text-decoration: none;
}

a:hover {
    text-decoration: underline;
}

form {
    background: #fff;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 1px 3px rgba(0,0,0,0.1);
}

div {
    margin-bottom: 10px;
}
```

Puede analizar el código de esta aplicación de ejemplo para obtener una idea de cómo implementar la funcionalidad requerida en el siguiente enlace: Código de la Aplicación de Inventario.

15 Actividad

Crea una aplicación web simple utilizando Ruby on Rails para gestionar un inventario. La aplicación debe permitir al usuario realizar las siguientes operaciones:

- Ver la lista de ítems en el inventario.
- Crear un nuevo ítem en el inventario.
- Ver los detalles de un ítem en el inventario.
- Editar un ítem en el inventario.
- Eliminar un ítem del inventario.

Utiliza los conceptos y pasos descritos anteriormente para crear la aplicación. Asegúrate de probar la funcionalidad de la aplicación y de mejorar el diseño con un poco de estilo CSS.

Ver Solución

```
# app/controllers/items_controller.rb

class ItemsController < ApplicationController
  def index
    @items = Item.all
  end

  def show
    @item = Item.find(params[:id])
  end

  def new
    @item = Item.new
  end

  def create
    @item = Item.new(item_params)
    if @item.save
      redirect_to @item
    else
      render :new
    end
  end

  def edit
    @item = Item.find(params[:id])
```

```
  end

  def update
    @item = Item.find(params[:id])
    if @item.update(item_params)
      redirect_to @item
    else
      render :edit
    end
  end

  def destroy
    @item = Item.find(params[:id])
    @item.destroy
    redirect_to items_path
  end

  private

  def item_params
    params.require(:item).permit(:name, :description, :quantity)
  end
end
```

```
# app/views/items/index.html.erb

<h1>Inventory</h1>

<%= link_to 'New Item', new_item_path %>

<ul>
  <% @items.each do |item| %>
    <li>
      <%= link_to item.name, item_path(item) %>
      <%= link_to 'Edit', edit_item_path(item) %>
      <%= link_to 'Destroy', item, method: :delete, data: { confirm: 'Are you sure?' } %>
    </li>
  <% end %>
</ul>
```

```
# app/views/items/new.html.erb

<h1>New Item</h1>

<%= form_with model: @item, local: true do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
```

```
  </div>
  <div>
    <%= form.label :description %>
    <%= form.text_area :description %>
  </div>
  <div>
    <%= form.label :quantity %>
    <%= form.number_field :quantity %>
  </div>
  <div>
    <%= form.submit %>
  </div>
<% end %>
```

```
# app/views/items/show.html.erb

<h1><%= @item.name %></h1>

<p><%= @item.description %></p>

<p>Quantity: <%= @item.quantity %></p>

<%= link_to 'Edit', edit_item_path(@item) %> |

<%= link_to 'Back', items_path %>
```

```
# app/views/items/edit.html.erb

<h1>Edit Item</h1>

<%= form_with model: @item, local: true do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>
  <div>
    <%= form.label :description %>
    <%= form.text_area :description %>
  </div>
  <div>
    <%= form.label :quantity %>
    <%= form.number_field :quantity %>
  </div>
  <div>
    <%= form.submit %>
  </div>
<% end %>
```

```
# app/assets/stylesheets/application.css

body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 20px;
  background-color: #f5f5f5;
}

h1 {
  color: #333;
}

ul {
  list-style: none;
  padding: 0;
}

li {
  padding: 10px;
  background: #fff;
  margin-bottom: 10px;
  border-radius: 5px;
  box-shadow: 0 1px 3px rgba(0,0,0,0.1);
}

a {
  color: #007bff;
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}

form {
  background: #fff;
  padding: 20px;
  border-radius: 5px;
  box-shadow: 0 1px 3px rgba(0,0,0,0.1);
}

div {
  margin-bottom: 10px;
}
```

16 Conclusión

En esta introducción a Ruby on Rails, hemos cubierto los conceptos básicos de Rails y hemos creado una aplicación web simple para gestionar un inventario. Rails es un framework poderoso y flexible que facilita el desarrollo de aplicaciones web. Con Rails, puedes crear aplicaciones web de manera rápida y eficiente, siguiendo las mejores prácticas de desarrollo web.