Objectives

- Working with String Arrays and Hashes
- Comparison and Logical Operators
- Conditional Statements
- Working with Loops

Working with String Arrays and Hashes

Strings

We use **strings** to store text values like names, messages, or any kind of written data. Strings are created by wrapping text in either double quotes (`"`) or single quotes (`'`):

```
name = "Alice"
greeting = 'Hello, world!'
```

But strings in Ruby aren't just plain text Ruby gives us **many built-in methods** to work with and manipulate strings easily.

Accessing Characters in a String

In Ruby, we can access individual characters in a string using **index numbers**. Just like arrays, string indices start at **0** so the first character is at position `0`, the second at `1`, and so on.

```
word = "Ruby"  puts word[0]  # => "R"
puts word[1]  # => "u" puts word[3]  # => "y"
```

If we try to access an index that's out of bounds, Ruby will return `nil`:

```
puts word[10]  # => nil
```

Negative Indexing

We can also use **negative numbers** to count from the end of the string:

```
word = "Ruby"  puts word[-1]  # => "y" (last character)
puts word[-2]  # => "b"
```

Accessing Substrings

We can get more than one character using a **range** or a starting index with length:
Range:

```
word = "Programming"
puts word[0..4]   # => "Progr"  (from index 0 to 4)
```

Start and length:

```
puts word[0, 5]   # => "Progr"  (start at 0, take 5 chars)
```

String Methods

`.length`

Returns the number of characters in the string:

```
name = "Alice"
puts name.length     # => 5
```

`.upcase` and `.downcase`

Convert a string to all **uppercase** or **lowercase**:

```
puts "hello".upcase   # => "HELLO"
puts "WORLD".downcase # => "world"
```

`.strip`

Removes extra whitespace from the beginning and end:

```
input = "   hello   "
puts input.strip      # => "hello"
```

`.include?`

Checks if a string **contains** a certain substring:

```
sentence = "Ruby is fun"
puts sentence.include?("fun")    # => true
puts sentence.include?("boring") # => false
```

`.concat` or `+`

Adds two strings together:

```
greeting = "Hello, " + "Alice"
puts greeting         # => "Hello, Alice"
```

You can also use interpolation:

```
name = "Bob"
puts "Hello, #{name}" # => "Hello, Bob"
```

`.gsub`

Replaces parts of a string:

```
text = "I like Java"
puts text.gsub("Java", "Ruby")  # => "I like Ruby"
```

`.split`

Splits a string into an array, based on a character or space:

```
words = "apple,banana,orange"
puts words.split(",")  # => ["apple", "banana", "orange"]
```

Arrays

We use **arrays** to store **ordered collections** of values. These values can be of the same type or different types strings, numbers, booleans, and even other arrays. Arrays are written with square brackets `[]`, and values are separated by commas:

```
colors = ["red", "green", "blue"]
numbers = [1, 2, 3, 4, 5]
```

```
mixed = ["hello", 42, true]
```

Accessing Elements in an Array

Just like strings, we access array elements using **index numbers**, starting from `0`:

```
colors = ["red", "green", "blue"]
puts colors[0]  # => "red"
puts colors[1]  # => "green"
puts colors[2]  # => "blue"
```

If we try to access an index that doesn't exist, Ruby returns `nil`:

```
puts colors[10]  # => nil
```

Negative Indexing

We can also access array elements from the end using **negative indices**:

```
colors = ["red", "green", "blue"]
puts colors[-1]  # => "blue" (last element)
puts colors[-2]  # => "green"
```

Accessing Subarrays

We can extract a portion of an array using a **range** or by giving a **start index and length**:
Range:

```
numbers = [10, 20, 30, 40, 50]
puts numbers[1..3]  # => [20, 30, 40]
```

Start and length:

```
puts numbers[0, 2]  # => [10, 20]
```

Array Methods

`.length`

Returns the number of elements in the array:

```
colors = ["red", "green", "blue"]
puts colors.length   # => 3
```

`.unshift(value)`

Inserts an element at the start.

```
colors = ["red", "green", "blue"]
colors.unshift("black") # => ["black", "red", "green", "blue"]
```

`.push` / `.<<`

Adds an element to the end:

```
colors = ["red", "green", "blue"]
colors.push("yellow")
colors << "purple"
# => ["red", "green", "blue", "yellow", "purple"]
```

`.insert(index, value)`

Adds an element at the given `index`.

```
colors = ["red", "green", "blue"]
colors.insert(2, "black")  # => ["red", "green", "black", "blue"]
```

`.shift`

Removes and returns the first element.

```
colors = ["red", "green", "blue"]
first_color = colors.shift
puts first_color      # => "red"
```

`.pop`

Removes and returns the last element:

```
colors = ["red", "green", "blue"]
last_color = colors.pop
puts last_color       # => "blue"
```

`.delete`

Removes all occurrences of `value`.

```
arr = [1, 2, 3, 2, 4]
arr.delete(2)  # => [1, 3, 4]
```

`.delete_at`

Removes the element at the given index.

```
arr = [1, 2, 3, 4]
arr.delete_at(1)  # => [1, 3, 4]
```

`.uniq!`

Removes duplicate values.

```
arr = [1, 2, 2, 3]
arr.uniq!  # => [1, 2, 3]
```

`.include?`

Checks if the array contains a value:

```
colors = ["red", "green", "blue"]
puts colors.include?("green")  # => true
```

`.reverse`

Returns a new array with the order reversed:

```
colors = ["red", "green", "blue"]
puts colors.reverse   # => ["blue", "green", "red"]
```

`.sort`

Sorts the array (if elements are comparable):

```
numbers = [3, 1, 5, 2]
puts numbers.sort     # => [1, 2, 3, 5]
```

`.join`

Combines array elements into a single string:

```
colors = ["red", "green", "blue"]
puts colors.join(", ")  # => "red, green, blue"
```

Hashes

We use **hashes** to store **key-value pairs** like a dictionary. Each key is linked to a specific value, making hashes perfect for representing things like settings, user profiles, and objects with properties.
Hashes are written using curly braces `{}` and use either strings or symbols as

```
person = {
"name" => "Alice",
"age" => 30
}
# Or using symbols (more common in Ruby):
person = {
name: "Alice",
age: 30
}
```

Accessing Hash Values

To access a value, use the corresponding key:

```
person = {
name: "Alice",
age: 30
}
```

```
puts person[:name]  # => "Alice"
puts person[:age]   # => 30`
```

If the key doesn't exist, Ruby returns `nil`:

```
person = {
name: "Alice",
age: 30
}
puts person[:email]  # => nil
```

Adding or Updating Entries

You can add new key-value pairs or update existing ones:

```
person = {
name: "Alice",
age: 30
}
person[:email] = "alice@example.com"
person[:age] = 31
```

We can use `.store` to add key value

```
person.store(:city, "London")  # => "London"
```

Hash Methods

`.keys` and `.values`

Get an array of all keys or values:

```
person = {
name: "Alice",
age: 30,
email: "alice@example.com"
}
puts person.keys   # => [:name, :age, :email]
puts person.values # => ["Alice", 31, "alice@example.com"]
```

`.delete(key)`

Removes a key-value pair:

```
person = {
name: "Alice",
age: 30,
email: "alice@example.com"
}
person.delete(:email)
```

`.clear`

Empties the entire hash.

```
person = {
name: "Alice",
age: 30,
email: "alice@example.com"
}
person.clear  # => {}
```

`.has_key?` / `.key?` / `.include?`

Check if a key exists:

```
person = {
name: "Alice",
age: 30,
email: "alice@example.com"
}
puts person.key?(:name)   # => true
puts person.include?(:age) # => true
```

`.has_value?`

Checks if a value exists.

```
person = {
name: "Alice",
```

```
age: 30,
email: "alice@example.com"
}
person.value?("Alice")  # => true
```

`.merge`

Combine two hashes. It returns a new hash (doesn't change the original):

```
defaults = { font: "Arial", size: 12 }
settings = { size: 14 }
puts defaults.merge(settings)  # => { font: "Arial", size: 14 }
```

Using Strings as Keys

While symbols are more common, you can also use strings as keys:

```
person = {
"name" => "Alice",
"age" => 30
}
puts person["name"]  # => "Alice"
```

`"name"` and `:name` are treated as **different keys**!

Comparison and Logical Operators

Comparison Operators

Comparison operators help us compare values or variables with each other and create conditions based on the results. These comparisons always return a **boolean value** — either `true` or `false`.
Ruby provides us with the following comparison operators:

- `>` (**Greater than**): Checks if the first operand is greater than the second.

```
5 > 4   # => true
3 > 10  # => false
```

- `<` (**Less than**): Checks if the first operand is less than the second.

```
4 < 5   # => true
7 < 3   # => false
```

- `==` (**Equal to**): Checks if both operands are equal.

```
4 == 4   # => true
"hi" == "hello"  # => false
```

- `!=` (**Not equal to**): Checks if the operands are not equal.

```
1 != 0   # => true
5 != 5   # => false
```

- `>=` (**Greater than or equal to**): Checks if the first operand is greater than or equal to the second.

```
2 >= 2   # => true
3 >= 5   # => false
```

- `<=` (**Less than or equal to**): Checks if the first operand is less than or equal to the second.

```
2 <= 3   # => true
4 <= 1   # => false
```

Logical Operators in Ruby

Ruby provides **logical operators** to combine multiple conditions and build more complex expressions. The result of a logical operation is always a **boolean value** — either `true` or `false`.

`||` (OR)

- Returns `true` if **at least one** of the conditions is true.
- Returns `false` only if **all** the conditions are false.

```
puts true || false   # => true
puts false || false  # => false
```

`&&` (AND)

- Returns `true` only if **all** the conditions are true.
- Returns `false` if **any** of the conditions is false.

```
puts true && true    # => true
puts true && false   # => false
```

`!` (NOT)

- Reverses the logical state of the condition.
- If the condition is `true`, `!` makes it `false`, and vice versa.

```
puts !true   # => false
puts !false  # => true`
```

Truth Table for Logical Operators:

A	B	A && B	A | B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Conditional Statements

Conditional statements are fundamental programming constructs that enable our code to make decisions and execute different actions based on specific conditions. They control program flow by evaluating whether certain criteria are met, allowing for dynamic behavior that responds to different situations, inputs, or values.

`if`, `else`, and `elsif` Statement

Single Condition with `if`

When we need to execute code only when a condition is true:

```
if condition
  # Code executes only if condition evaluates to true
end
```

Example:

```
age = 18

if age >= 18
  puts "You are an adult."
end
# Output when age is 18: "This person is an adult."
```

Alternative Path with `if-else`

When we want to handle both true and false cases differently:

```
if condition
  # Code for true case
else
  # Code for false case
end
```

Example

```
is_raining = false

if is_raining
  puts "Bring an umbrella."
else
  puts "No umbrella needed today."
end
# Output: "No umbrella needed today."
```

Multiple Conditions with `elsif`

When we need to check several possible conditions in sequence:

```
if condition1
  # First case
elsif condition2
  # Second case
else
  # Default case
end
```

Example:

```
score = 85
if score >= 90
  puts "Excellent performance"
elsif score >= 80
  puts "Good job"
else
  puts "Room for improvement"
end
# Output: "Good job"
```

The `case` Statement

`case` is Ruby's version of `switch` found in other languages. It's cleaner when comparing the same value against multiple options.

```
case variable
when value1
  # Code for value1
when value2
  # Code for value2
else
  # Runs if no match
end
```

Example:

```
day = "Monday"

case day
```

```
when "Monday"
  puts "Weekday 1"
when "Tuesday"
  puts "Weekday 2"
when "Saturday", "Sunday"
  puts "Weekend!"
else
  puts "Unknown day"
end
```

Loops

Loops help us **repeat code** multiple times without rewriting it. They're essential when we want to **perform the same task** on a group of values, or when we don't know in advance how many times we need to run some code.
Ruby gives us several types of loops, and each is useful in different situations.

`while` Loop

A `while` loop keeps running **as long as a condition is true**.

```
count = 1
while count <= 5
    puts "Count is #{count}"
    count += 1
end
```

Output:

```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
```

If the condition is `false` to begin with, the code inside won't run at all.

`until` Loop

`until` is the **opposite** of `while`: it runs the loop until the condition becomes true.

```
count = 1
until count > 5
    puts "Count is #{count}"
    count += 1
end
```

Same output as above, but using `until`.

`for` Loop

A `for` loop goes through a **range or a collection** like an array, we use it when we know how much time we want to repeat a statement:

```
for i in 1..3
    puts "Iteration #{i}"
end
```

We can also use it with arrays:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits
    puts fruit
end
```

`.each` Loop

This is the **most common and Ruby-ish** way to loop through arrays or hashes.

Array Example:

```
colors = ["red", "green", "blue"]

colors.each do |color|
  puts "Color: #{color}"
end
```

Hash Example:

```ruby
person = { name: "Alice", age: 30 }

person.each do |key, value|
  puts "#{key}: #{value}"
end
```

`loop do` and `break`

`loop do` creates an **infinite loop**, unless we explicitly stop it with `break`.

```ruby
counter = 1

loop do
  puts "Looping: #{counter}"
  counter += 1
  break if counter > 3
end
```

`next` and `break` Keywords

- `break` : Stops the loop entirely.
- `next` : Skips the rest of the loop for the current iteration and jumps to the next one.

Example with next

```ruby
for i in 1..5
  next if i == 3
  puts i
end
# Skips 3
```

Example with break

```ruby
for i in 1..5
  break if i == 4
  puts i
end
# Stops after 3
```

Tasks

Task 1

- Create string variable with name `word` and value `"Developer"`
  - Print the first and last characters using indexing`
  - Convert to uppercase and print`
- Create string variable named `phrase` with this text `"I love Python"`
  - Replace "Python" with "Ruby" and print the result
- Create string variable named `sentence` with this text `"Learning Ruby is fun"`
  - Print true if "Ruby" is in the sentence
  - Print the number of characters in the sentence`

Task 2

- Create an array of colors: `["red", "green", "blue"]`
  - print last and first element
  - Add "yellow" to the end of the array
  - Insert "black" at the beginning
  - Remove and print the last element
  - Check if "green" exists in the array
- Create a new array with only unique values from `[1, 2, 2, 3, 4, 4, 5]`
  - Remove the second item
  - Insert 25 at index 2`
- Sort the array `[5, 3, 8, 1]` in ascending order

Task 3

- Create a hash representing a person with name, age, and city
  - Print the name
    - Add a new key-value pair for occupation
    - Print all the keys of the hash
  - Check if the hash contains a key :age
  - Merge with another hash `{ hobbies: ["reading", "coding"] }`
  - Delete the city information from the hash

Task 4

- Write an if-else statement that checks if a number is even or odd

- Create a grading system using elsif (A for >=90, B for >=80, etc.)
- Convert the grading system to a case statement
- Write a program that recommends clothing based on temperature:
  - Above 30°C: "Wear shorts"
  - 20-30°C: "T-shirt weather"
  - Below 20°C: "Bring a jacket"

Task5

Write a C program that reads three numbers, **a**, **b**, and **c**, from the user, and then solves the quadratic equation:

$$a^2 + bx + c = 0$$

The program should handle the following cases:

1. When the discriminant b2−4ac is positive, there are two real and distinct roots.
2. When the discriminant is zero, there is one real root .
3. When the discriminant is negative, the equation has no real solutions.

Task 6

- Create a program that asks the user to enter a number and calculates its factorial using a `while` loop.
- Ask the user for two numbers and display all prime numbers between them using a `for` loop.
- Ask the user to enter a number `n`, and use an `until` loop to print all even numbers from 0 up to `n`.