Ruby :

Ruby is a readable and developer-friendly language that's easy to understand. In Ruby, we don't need to declare data types explicitly—we just assign values to variables, and the language automatically understands the type.

Ruby Syntax:

1. **No need to declare data types**

   Ex: name = “soundarya”
   Numbers = 1234
   Is_admin = true

2. No semicolons required:
   Name: “Soundarya”

3. Comments start with #

**What is Indentation?**

**Indentation = spaces at the beginning of a line of code** to show that it's **inside** something else.

```
def test
  puts "Hello, world!"
end
```

Here we added the 2 spaces after puts word because we are informing that the line is inside the method.

Problem:

```
n = 10  # upper limit
m = 3   # multiple value

(1..n).each do |num|
  if num % m == 0
    puts "This number #{num} is a multiple of #{m}"
  else
    puts num
  end
end
```

## Data Types:

1. **Numbers :**

   **Numbers** in Ruby are just digits.
   We can declare num = 12;
   Number = 5.3

2. **String :**

   A **String** is a group of letters, words, or sentences enclosed in “ ”

   ```
   name = "Soundarya"
   city = 'Kadapa'
   ```

3. **Booleans:**

   ```
   is_login = true
   ```

4. **Arrays: The index starts from 0.**

   ```
   fruits = ["apple", "banana", "mango"]
   numbers = [1, 2, 3, 4, 5]
   ```

5. **Hashes: Storing all the information in one {key:value}**

   ```
   person = { name: "Soundarya", age: 22 }

   Accessing:
   puts person[:name]     # → Soundarya
   puts person[:age]      # → 22

   To update the values:
   person[:name] = "Soundarya Reddy"
   person[:age] = 23
   ```

6. **Symbols:**
   **{:name}**

   ```
   person = {
     name: "Soundarya",
     age: 22
   }

   puts person[:name]   # → Soundarya
   ```

That `name:` is a shortcut for `:name =>`. (Old version)

In Method Arguments (keyword args)

```
def greet(name:, city:)
  puts "Hello #{name} from #{city}"
end

greet(name: "Soundarya", city: "Kadapa")
```

**Problem Statement:**

```
# Step 1: Creating an array with different data types
mixed_array = [1, "hello", :symbol, true, 3.14, false, "world",
10, :ruby]

# Step 2: Defining a method to count each data type

def count_types(array)
  result = {}

  array.each do |item|
    type = item.class
     result[type] = 0
     result[type] += 1
  end

  result
end

# Step 3: Calling the method and printing the result
puts count_types(mixed_array)
```

## Variables:

In Ruby, we don't need to declare the variable type. We can simply assign a value to a variable, and Ruby will understand the type automatically.
However, we cannot declare a variable without assigning a value.

```
name = "Soundarya"
age = 22
is_happy = true
```

**Constants:**

We have to declare the variables with Capital letters, and when we reuse the same variable name, it throws a warning message.

To see the warning message:

```
# First declaration
NAME= "Hello, Soundarya!"
puts NAME

# Second declaration (same constant)
NAME= "Hi again!"
puts NAME
```

**Scope :**

Local variables
Global variables
Instance variables
Class variables
Constant variables

1.Local variables are only accessible within the block, method, or class where they are defined. They begin with a lowercase letter or _

Ex: def greet

```
  message = "Hello"  # local variable

puts message
```

```
  end

 greet

# Output: Hello
```

2. Global variables are accessible from anywhere in the Ruby program. They begin with a $.

```
$global_message = "Hello from anywhere!"

def show_message

 puts $global_message

end

show_message

puts $global_message

# Output:

# Hello from anywhere!

# Hello from anywhere!
```

3. Instance variables are tied to a specific object. They begin with @ and are available across methods within the same object.

```
class Person

  def initialize(name)

  @name = name   # instance variable

  end


   def show_name

  puts "Name is #{@name}"      # access instance variable
```

```
   puts "Current object is #{self}" # access object using self
  end
End
```

4.Class Variables: A **class variable** is shared **across the class and all its objects**.

It **starts with** **@@**

**5.Constant Variables:** A constant is a variable that should not change once assigned.
It starts with an uppercase letter, usually ALL_CAPS by convention.
Accessible across the class or module where it's defined.

**Problem:**

```
PI = 3.14         # Constant
$greet = "Hi!"    # Global variable

def show_scope
  local = "I'm local"   # Local variable
  puts PI
  puts $greet
  puts local
end
show_scope
puts PI
```

```
puts $greet

puts local  #  Error: local is not accessible here
```

## Control Flow:

**If Condition**: The code run when the condition is true

```
Num = 10

Number = 20
```

```
Ex: If num > number

Puts “#{num} is greater than #{number}”

Else

Puts “#{number} is greater than #{num}”
```

1. **Unless Condition:**

Opposite of if. Runs code **only if** the condition is **false**.

```
logged_in = false

unless logged_in
  puts "Please log in"
else
  puts "Welcome back!"
End
```

2. **Case:**

```
The case will check multiple possible conditions

grade = "B"
```

```
case grade
when "A"
  puts "Excellent"
when "B"
  puts "Good job"
when "C"
  puts "Keep trying"
else
  puts "Invalid grade"
end
```

### 3. While Loop:

The code will repeat until the condition true

```
 i = 1
while i <= 5
  puts i
  i += 1
end
```

### 4. Until:

The code will repeat until the condition is true

```
i= 1
```

```
Until i> 5

Puts i

i+=1

end
```

5. For loop: a range of collection

```
   for i in 1..3

     puts "Hello #{i}"

   end
```

6. Loop Do: infinite loop until we break

```
x = 0

loop do

  x += 1

  puts x

  break if x == 5

End
```

**Problem statement:**

**If/Else**

**Score = 35**

```
  if score >= 35

    puts "Pass"

  else
```

```
    puts "Fail"
  End
```

**Case :**

```
  case score
  when 0..34
    puts "Fail"
  when 35..100
    puts "Pass"
  else
    puts "Invalid score"
  end
End
```

## Methods:

a **method** is a block of code that performs a task. It starts with def and ends with end.

Ex: Without parameters

```
def greet
  puts "Hello!"
end
```

Greet

Ex: Default parameters:

```
def greet(name = "Guest")
  puts "Hello, #{name}!"
end

greet("Soundarya")   # Output: Hello, Soundarya!
greet                # Output: Hello, Guest!
```

**Method with parameters:**

```
def greet(name)
  puts "Hello, #{name}!"
end
greet("Soundarya")   # Output: Hello, Soundarya!
```

**Splat Operator (*):** The splat operator allows to handle multiple parameters as a group (array)

```
Ex: def list_names(*names)
  puts names
end
list_names("Ram", "Seetha", "Lakshman")
```

Problem statement:

```
def greet_user(name, gender = 'female')
```

```
  if gender == 'male'
    puts "Mr. #{name}, welcome to Freshworks"
  else
    puts "Ms. #{name}, welcome to Freshworks"
  end
end

users = {
  "Ravi" => "male",
  "Anu" => "female",
  "Kiran" => nil
}

users.each do |name, gender|
  greet_user(name, gender)
end
```

**File reading :**

```
# Read names from input file
input_file = "names.json"
output_file = "capitalized_names.json"
```

```
# Read each line, capitalize the name, and collect in an array

capitalized_names = []


file = File.read('data.json')

data = JSON.parse(file)


# Loop through the JSON data

data.each do |item|

  puts item

End
```

## Read Files:

```
require 'json'

require 'csv'


# 1. Read a plain text file (.txt)

file_path = “file.txt”

def read_text_file(file_path)

  File.readlines(file_path).each { |line| puts line.strip }

end
```

## Blocks, Procs, Lambdas:

1. Blocks:
   A **block** in Ruby is a chunk of code enclosed between do...end or

{} that can be **passed to a method** and **executed** from inside that method using the yield keyword or &block.call

## Ex:

```
def green

Yield        // calling the method to print

end

green do  // block

Puts “Soundarya”

end
```

## With block.call:

**Ex:**

```
def greet(&block)

     block.call    # this runs the block

end

greet do         ## block

  puts "Hello from block"

end
```

## Proc:

A **Proc** is an object in Ruby that stores a block of code which you can reuse and call later.

Ex: green = proc.new{puts=”Soundarya Bhavanasi”}

green.call

Lampda: Same as Proc but the difference is it will not accept the without arguments null values. Where

```
Ex: green = proc.new{|name|puts=”Soundarya Bhavanasi #{name}”}

green.call

// here, without giving value, we can run passing null values.


For mathematical things it will throw an error


Ex: add_proc = Proc.new { |a, b| puts a + b }
  add_proc.call(1)
# => NoMethodError (undefined method `+` for nil)


Ex: green =lambda.new{|name|puts=”Soundarya Bhavanasi #{name}”}

Lambda.call

// Here it throws the argument saying argument missing


Problem statement:


add_proc = Proc.new { |a, b|

puts "Add #{a + b}"

puts "Add #{a - b}"

puts "Add #{a * b}"

puts "Add #{a / b}"

}

add_proc.call(2,4)

calc_lambda = lambda {|a, b|
```

```
  puts "Add: #{a + b}"

  puts "Subtract: #{a - b}"

  puts "Multiply: #{a * b}"

  puts "Divide: #{a / b.to_f}"

}


calc_lambda.call(2, 4)
```

### Begin-rescue-ensure:

Run code that might raise an error (begin)
Handle the error gracefully (rescue)
Always run cleanup code, no matter what (ensure)

```
def divide(a, b)

  begin

    result = a / b

  rescue errormessage => e

    puts "Error: #{e.message}"

  ensure

  end

end


puts divide(10, 0)
```

**Custom exceptions:** Are used to **create our own error messages** and handle **specific types of errors** that may occur while running the code.

## Object-Oriented Concepts

### 1. Class

A **class** is like a **blueprint** or **template** to create objects.

Ex: class Car

```
  def drive

    puts "The car is driving"

  end

End
```

```
my_car = Car.new  # creating an object

my_car.drive       # Output: The car is driving
```

### 2. Module:

A module is a container for methods and constants.

We can use it whenever we want by extending with the class name.

3. **Inheritance:**
When one class gets the **features (methods/variables)** of another class.

4. **mixins**

Mixins happen when you **include modules inside a class**.
Ruby doesn't support **multiple inheritance**, but **you can use many modules** → This is called a **mixin**.

5. **Access Modifiers:**

**publc** Anyone can call
(default in Ruby)

**privae** Can be used only inside
the class

**protectd** Can be used inside class
or subclasses