

# CSE/ISE 337: Scripting Languages Lecture 9


Prof. Abid M. Malik  
Lecture 2/19/2024

1

## Today's Agenda

- Ruby Basics
- Input/output
- Data Types and Objects
- Variables
- Ruby Syntax
- Ruby Keywords
- Strings the main data type
- Expression and Operators
- Statement Control statements

2

PYTHON VS. RUBY		
PYTHON		RUBY
was Created in 1991 by Guido Van Rossum	LANGUAGE	was Created in 1995 by Yukihiro Matsumo
A diverse community with extensive ties to Linux and academia	PROS	Tons of features out of the box for web development
Often very explicit and inelegant to read	CONS	Can be tough to debug at times. Very web-focused
Django	WEB FRAMEWORKS	Ruby on Rails
Very stable and diverse but innovate slower	COMMUNITY	Innovate quicker but causes more things to break
Google YouTube Dropbox The Washington Post	USAGE	Twitter Airbnb Github Shopify

3

## Introduction to Ruby Programming

- Ruby is a popular scripting language(developed by Yukihiro Matsumoto)
- Ruby is a general purpose language
- Active and supportive community
- Rich ecosystem of APIs and frameworks (e.g., Ruby on Rails)
- Ruby has wide commercial use (e.g., Airbnb, GitHub, Hulu)



4

# General Characteristics of Ruby

- Ruby is object-oriented
  - Everything is an object
  - Object manipulations result in more objects
- Ruby is dynamically typed
  - Explicit type declarations are not required
  - Characteristics of a scripting language

5

## Getting Started With Ruby

- Mac OSX
  - Use a package manager
  - Homebrew is commonly used
    - Install from <https://brew.sh/>
      - If you have brew
        - \$ brew update (not required if already updated)
        - \$ brew install ruby
    - If brew update fails, then follow instructions at <https://docs.brew.sh/Common-Issues>
- Linux
  - apt (Debian or Ubuntu)
    - \$ sudo apt-get install ruby-full
  - yum (Fedora, RHEL)
    - \$ sudo yum install ruby
  - Make sure to run apt update or yum update if not current
- Windows
  - Download and run the Ruby installer <https://rubyinstaller.org/>



6

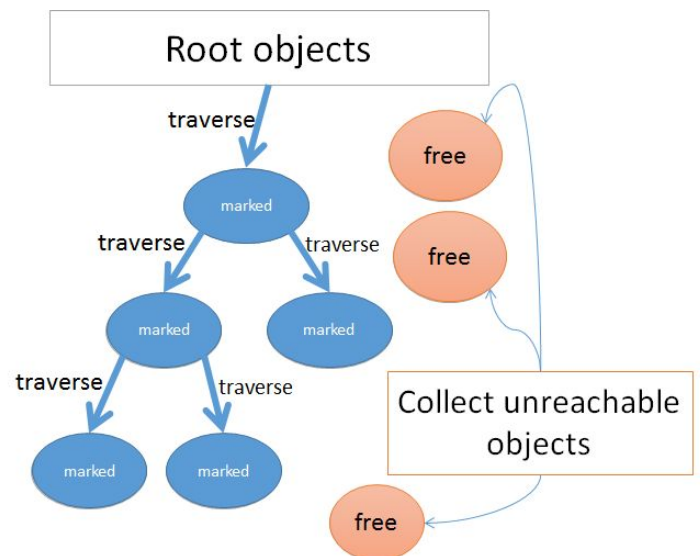
# Using Ruby

- Configure path/to/ruby in environment variable PATH to seamlessly run from command line
- Start Interactive Shell from command line  
`$ irb`
- Run Ruby programs from command line  
`$ ruby /path/to/program.rb`
- Use text editors to write Ruby programs
  - Atom, Notepad++, Emacs
- Use any IDE that supports Ruby
  - RubyMine, IntelliJ, Netbeans

7

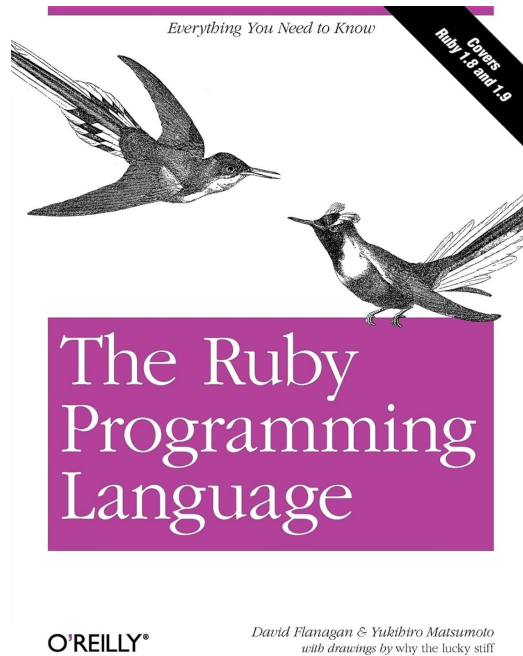
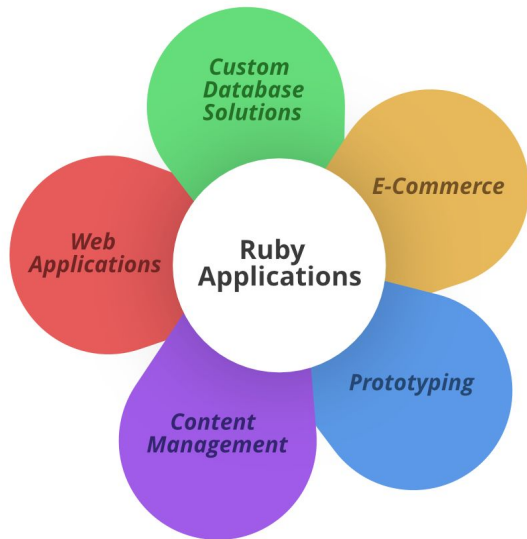
## Garbage Collection

Ruby has a built in garbage collector which automatically destroys objects and reclaims the memory they used when they are no longer reference in your program



8

# Ruby Applications



9

## Input/Output from the User

- `print` command
- `puts` command
- `gets`
- `gets.chomp`
- `gets.chop`
- `gets.strip`

10

# Variables

- Variable names must begin with a letter (a-z or A-Z) or an underscore (\_).
- After the first character, variable names can include letters, numbers, and underscores.
- Ruby is case-sensitive, so `my_variable` and `My_Variable` are treated as two different variables.
- Variable names cannot be reserved keywords like `if`, `else`, `end`, `class`, `module`, etc.

11

# Variables

- Global variables
- Class variables
- Instance variables
- Local variables

12

# Variables

Symbol	Type of Variable
[a-z] or _	Local Variable
@	Instance Variable
@@	Class Variable
\$	Global Variable

13

## Evaluating a variable

- In Ruby, you can evaluate variables within a string using string interpolation or concatenation.
- String interpolation allows you to embed the value of a variable directly within a string.
  - **Note:** When using concatenation, you may need to convert non-string variables to strings using methods like `to_s`. However, string interpolation automatically converts variables to strings, so you don't need to explicitly call `to_s` on variables in that case.

14

# Local Variables

- A local variable name always starts with a lowercase letter(a-z) or underscore (\_).
- These variables are local to the code construct in which they are declared.
- A local variable is only accessible within the block of its initialization.
- Local variables are not available outside the method. There is no need to initialize the local variables.

15

## Local Variable shadowing

If you define a local variable with the same name as an outer scope variable, the outer scope variable will be temporarily "shadowed" or hidden within the inner scope.

Question: Does local variable shadowing exist in Python?

16



## Instance Variables:

- An **instance variable** name always starts with a @ sign.
- They are similar to Class variables but their values are local to specific instances of an object.
- Instance variables are available across methods for any specified instance or object i.e. instance variables can change from object to object.
- There is no need to initialize the instance variables and uninitialized instance variable always contains a nil value.

17

## Class variables

- A class variable name always starts with @@ sign. It is available across different objects.
- A class variable belongs to the class and it is a characteristic of a class.
- **They need to be initialized before use.** Another way of thinking about class variables is as global variables within the context of a single class.
- A class variable is shared by all the descendants of the class. An uninitialized class variable will result in an error.

18

# Global Variables

- A global variable name always starts with \$.
- Class variables are not available across classes.
- If you want to have a single variable, which is available across classes, you need to define a global variable.
- Its scope is global, means it can be accessed from anywhere in a program.
- By default, an uninitialized global variable has a *nil* value and its use can cause the programs to be cryptic and complex.

19

# Predefined Variables

- Many predefined variables:
  - In Ruby, there are several predefined variables that have special meaning and are available for use within your code. These variables provide information about the Ruby environment, current execution context, and other useful data.
- \_\_FILE\_\_
- \_\_LINE\_\_

20

# Syntax of Ruby

- The syntax of Ruby is designed to be intuitive and expressive, emphasizing readability and simplicity. Here are some key aspects of Ruby syntax:
  - End-of-line Termination: Unlike some other languages, Ruby does not require semicolons at the end of each line. Statements are typically terminated by the end of the line.
  - Blocks and Control Structures: Ruby uses do and end or curly braces {} to define blocks of code. Control structures such as if, else, elsif, unless, while, until, for, and case also use end to denote the end of the block.
  - Variable Naming: Variable names in Ruby are case-sensitive and can contain letters, numbers, underscores, and some special characters. Conventionally, variable names use snake\_case (e.g., my\_variable) and constants use SCREAMING\_SNAKE\_CASE (e.g., MY\_CONSTANT).

21

## Comments in Ruby

Single-line comments start with #, and multi-line comments are enclosed between =begin and =end.

22

# Keywords

BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	__FILE__
defined?	module	super	__LINE__

23

## Data Types in Ruby

- **Numbers/Numeric**

- a. Integer
- b. Real

- **Boolean**

- **Strings**

- **Hashes**

- **Arrays**

- **Symbols**

24

## Number/Numeric : Integer and float (real)

```
# Ruby program to illustrate the
# Numbers Data Type

# float type
distance = 0.1

# both integer and float type
time = 9.87 / 3600
speed = distance / time
puts "The average speed of a sprinter is #{speed} km/h"
```

25

## Number/Numeric : Integer and float (real)

- When performing arithmetic operations involving both integers and floats (real numbers), the result will typically be automatically converted to a float if any of the operands is a float. This process is known as "upcasting" or "promotion."
- Similar to Python!

26

# Strings

In Ruby, *string* is a sequence of one or more characters. It may consist of numbers, letters, or symbols.

Here strings are the objects, and apart from other languages, strings are mutable, i.e. strings can be changed in place instead of creating new strings.

String's object holds and manipulates an arbitrary sequence of the bytes that commonly represents a sequence of characters.

**Creating Strings:** To create the string, just put the sequence of characters either in double quotes or single quotes. Also, the user can store the string into some variable.

27

## String Interpolation

- String interpolation in Ruby is a feature that allows you to embed expressions (variables, method calls, etc.) directly within a string literal.
- When a string containing interpolation is evaluated, the expressions within the `#{}`  placeholders are replaced with their corresponding values.

28

# String Interpolation

- The only difference between using single and double quotes is that the double quotes will interpolate the variables but single quotes can't interpolate.

29

## Strings are objects:

- Ruby is an object-oriented language so string in Ruby are objects. Basically, an object is a combination of the data and methods which enhance the communication property.

30

## Access String Elements

User can access the string elements by using the *square brackets []*. In square brackets [], the user can pass the strings, ranges or indexes.

### Syntax:

```
name_of_string_variable[arguments]
```

31

## Creating Multiline Strings

In Ruby, a user can create the multiline strings easily whereas in other programming languages creating multiline strings requires a lot of efforts. There are three ways to create multiline strings in Ruby as follows:

1. **Using Double Quotes("")** It is the simplest way to create the multiline strings by just putting the string between the quotes. Between double quotes, the user can add the newline character and so on.
2. **Using (%/ /)** To create the multiline string just put the string between the %/ and /.
3. **Using (<< STRING STRING)** To create the multiline string just put the string between the << STRING and STRING. Here STRING should be in capital letters.

32



# String Replication

Sometimes a user may require to repeat some sort of string multiple times. So to make the replication of string in Ruby, make the use of (\*) operator. This operator is preceded by the string to be replicated and followed by the number of times to make replicas.

## Syntax:

```
string_variable_or_string * number_of_times
```

33

# Converting Strings to Numbers

- Ruby provides the `to_i` and `to_f` methods to convert strings to numbers. `to_i` converts a string to an integer, and `to_f` converts a string to a float.
- Ruby offers another way to perform this conversion. You can use the `Integer` and `Float` methods

34

## Converting Data to String

- Ruby provides the `to_s` method to convert any other type to a string:

35

## String Concatenation and Freeze

- `+`
- `<<`
- `.concat`
- `.freeze`
  - you cannot directly unfreeze the original string. Once a string is frozen in Ruby, it remains frozen and immutable for the lifetime of the program. This behavior is by design in Ruby.

36

## Inserting a string in a string

In Ruby, the `.insert` method is used to insert a substring into a string at a specified index position. It modifies the original string and returns the modified string.

The syntax for the `.insert` method is:

```
string.insert(index, substring)
```

37

## The `chomp` and `chop` methods

- The **`chomp` method** is used to remove trailing newline characters (`\n`) from the end of strings.
  - a. It can optionally remove other specified trailing characters.
  - b. If no argument is provided, `chomp` removes the trailing newline character by default.
- The **`chop` method** removes the last character from the end of a string.
  - a. It always removes exactly one character, regardless of its value.

38

## Case conversion for string

- `.downcase`
- `.upcase`
- `.swapcase`
- `.delete`
- `.ljust`
- `.rjust`

39

## Ruby Cheat Sheet

- [Ruby cheat sheet](#)

In Ruby, the ``#`` symbol is often used as a convention to denote instance methods when discussing code or documentation. For example, when referring to the ``to_s`` method of a string object, you might see it written as ``String#to_s``.

40

# Arithmetic Operators

- **Addition(+)**: operator adds two operands. For example, `x+y`.
- **Subtraction(-)**: operator subtracts two operands. For example, `x-y`.
- **Multiplication(\*)**: operator multiplies two operands. For example, `x*y`.
- **Division(/)**: operator divides the first operand by the second. For example, `x/y`.
- **Modulus(%)**: operator returns the remainder when first operand is divided by the second. For example, `x%y`.
- **Exponent(\*\*)**: operator returns exponential(power) of the operands. For example, `x**y`.

41

# Comparison Operators

- **Equal To(==)** operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false. For example, `5==5` will return true.
- **Not Equal To(!=)** operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false. It is the exact boolean complement of the `'=='` operator. For example, `5!=5` will return false.
- **Greater Than(>)** operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false. For example, `6>5` will return true.
- **Less than(<)** operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false. For example, `6<5` will return false.
- **Greater Than Equal To(>=)** operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, `5>=5` will return true.
- **Less Than Equal To(<=)** operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise it returns false. For example, `5<=5` will also return true.
- **Combined combination (<=>)** operator return 0 when first operand equal to second, return 1 when first operand is greater than second operand, and return -1 when first operand is less than second operand.
- **'eq1?'** This operator returns true if the receiver and argument have both the same type and equal values.
- **'Equal?'** This operator Returns true if the receiver and argument have the same object id.

42

# Assignment Operators

- **Simple Assignment (=)**: operator is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
- **Add AND Assignment (+=)** operator is used for adding left operand with right operand and then assigning it to variable on the left.
- **Subtract AND Assignment (-=)** operator is used for subtracting left operand with right operand and then assigning it to variable on the left.
- **Multiply AND Assignment (\*=)** operator is used for multiplying left operand with right operand and then assigning it to variable on the left.
- **Divide AND Assignment (/=)** operator is used for dividing left operand with right operand and then assigning it to variable on the left.
- **Modulus AND Assignment (%=)** operator is used for assigning modulo of left operand with right operand and then assigning it to variable on the left.
- **Exponent AND Assignment (\*\*=)** operator is used for raising power of left operand to right operand and assigning it to variable on the left.

43

# Bitwise Operators

- **Bitwise AND (&)** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- **Bitwise OR (|)** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
- **Bitwise XOR (^)** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- **Left Shift (<<)** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- **Right Shift (>>)** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- **Ones Complement (~)** This operator takes a single number and used to perform complement operation of 8-bit.

44

## Ternary Operator

It is a conditional operator which is a shorthand version of the if-else statement. It has three operands and hence the name ternary. It will return one of two values depending on the value of a Boolean expression.

### Syntax :

```
condition ? first_expression : second_expression
```

45

## Range Operators

Range operators are used for creating the specified sequence range of specified elements. There are two range operators in Ruby as follows:

- **Double Dot (..)** operator is used to create a specified sequence range in which both the starting and ending element will be inclusive. For example, **7 .. 10** will create a sequence like **7, 8, 9, 10**.
- **Triple Dot (...)** operator is used to create a specified sequence range in which only starting element will be inclusive and ending element will be exclusive. For example, **7 ... 10** will create a sequence like **7, 8, 9**.

46

# defined? Operator

The `defined?` the operator is a special operator which is used to check whether the passed expression is defined or not. It returns `nil` if passed argument is not defined, otherwise, it returns a string of that argument which defines that.

## Syntax:

```
defined? expression_to_be_checked
```

# Operator Precedence in Ruby

Method	Operator	Description
Yes	[ ] [ ] =	Element reference, element set
Yes	**	Exponentiation (raise to the power)
Yes	! ~ + -	Not, complement, unary plus and minus (method names for the last two are <code>+@</code> and <code>-@</code> )
Yes	* / %	Multiply, divide, and modulo
Yes	+ -	Addition and subtraction
Yes	>> <<	Right and left bitwise shift
Yes	&	Bitwise 'AND'
Yes	^	Bitwise exclusive 'OR' and regular 'OR'
Yes	<= < > >=	Comparison operators
Yes	<=> == === != =~ !~	Equality and pattern match operators ( <code>!=</code> and <code>!~</code> may not be defined as methods)
	&&	Logical 'AND'
		Logical 'AND'
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= { /= -= +=  = &= >>= <<= *= &&=    = **=	Assignment
	defined?	Check if specified symbol defined
	not	Logical negation
	or and	Logical composition
	if unless while until	Expression modifiers
	begin/end	Block expression



# Control Statements

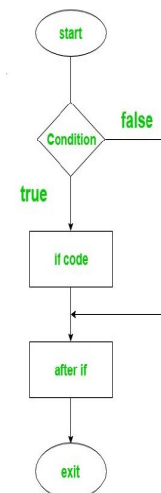
- `if` , `if-else`, `if-elsif`
- Ternary statement
- Loops
- Control Flow Alteration
- Break and Next statement
- redo and retry
- BEGIN and END Blocks

49

## if statement

Syntax:

```
if (condition)  
    # statements to be executed  
end
```

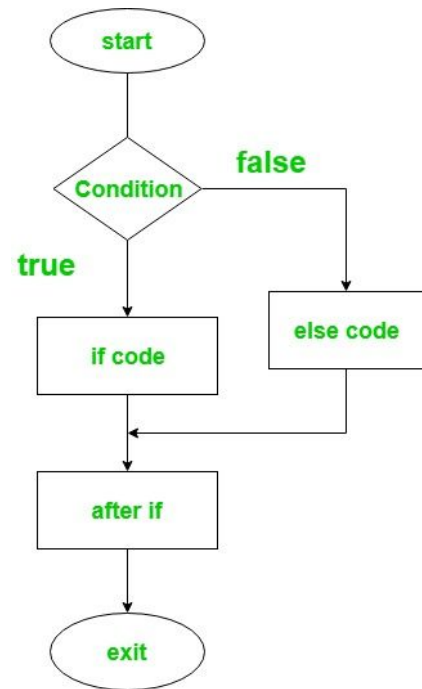


50

## if -else statement

Syntax:

```
if(condition)
    # code if the condition is true
else
    # code if the condition is false
end
```

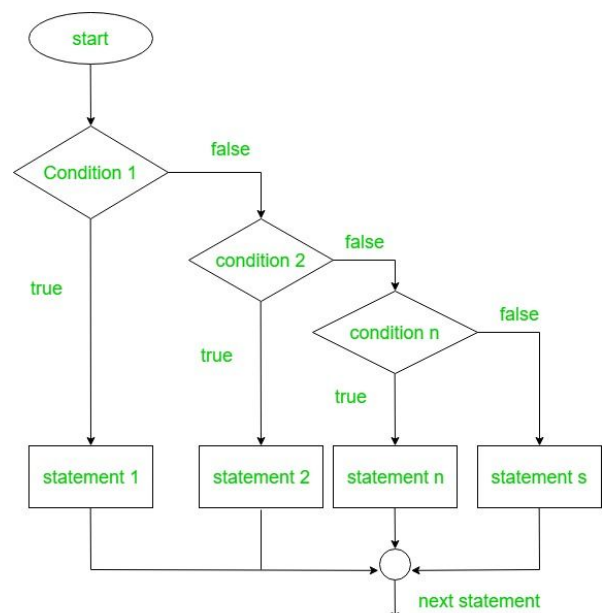


51

## If-elseif-else ladder statement

Syntax:

```
if(condition1)
    # code to be executed if condition1 is true
elseif(condition2)
    # code to be executed if condition2 is true
else(condition3)
    # code to be executed if condition3 is true
end
```



52

# Ternary Statement

Syntax:

```
test-expression ? if-true-expression : if-false-expression
```

53

## Loops in Ruby

- while loop
- for loop
- do..while loop
- until loop

54

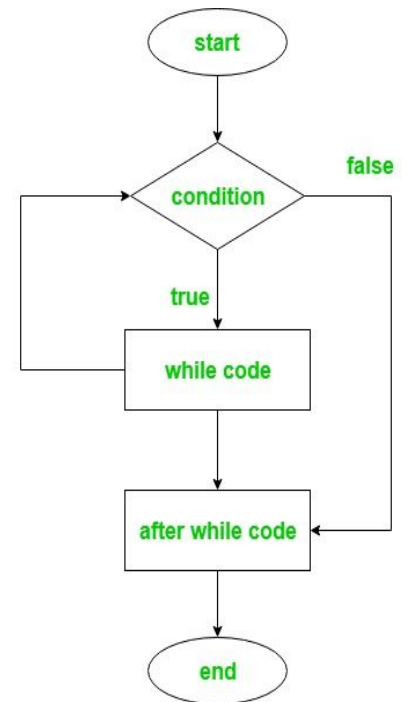
# While Loop

```
while conditional [do]
```

```
    # code to be executed
```

```
end
```

Note: A while loop's conditional is separated from code by the reserved word *do*, a *newline*, *backslash(\)*, or a *semicolon(;)*.



55

# for Loop

```
for variable_name[, variable...] in expression [do]
```

```
    # code to be executed
```

```
end
```

56

## do ... while Loop

*do while loop* is similar to while loop with the only difference that it checks the condition after executing the statements, i.e it will execute the loop body one time for sure. It is a **Exit-Controlled** loop because it tests the condition which presents at the end of the loop body.

Syntax:

```
loop do
```

```
  # code to be executed
```

```
break if Boolean_Expression
```

```
end
```

57

## until .. Loop

Ruby *until loop* will executes the statements or code till the given condition evaluates to true. Basically it's just opposite to the while loop which executes until the given condition evaluates to false.

Syntax:

```
until conditional [do]
```

```
  # code to be executed
```

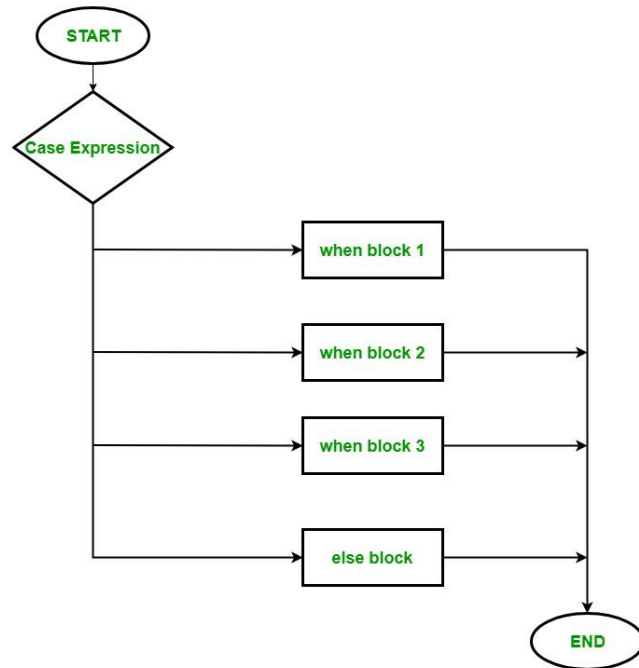
```
end
```

58

# Ruby case statement

Syntax:

```
case expression
when expression 1
  # your code
when expression 2
  # your code
.
.
else
  # your code
end
```



59

# Break Statement

Syntax:

```
break
```

60

## next Statement

Syntax:

**next**

61

## redo Statement

Syntax:

**redo**

*The redo statement is used to restart the current iteration of a loop or the iterator. There is a difference between the redo and next statement. next statement always transfers the control to the end of the loop where the statement after the loop can start to execute, but redo statement transfer the control **back to the top of block** or loop so that iteration can start over.*

62

## Other control statements

- `retry` statement (deprecated)
- `return` statement
- `catch and throw` statement

63

## BEGIN and END Blocks In Ruby

- Every Ruby source file can run as the **BEGIN** blocks when the file is being loaded and runs the **END** blocks after the program has finished executing.
- The BEGIN and END statements are different from each other. A program may contain multiple BEGIN and END blocks.
- If there is more than one BEGIN statement in a program, they are executed in the order. If there is more than one END statement, they are executed in the reverse of the order. the first END one is executed last.
- An open curly brace always come after BEGIN and END keyword

64



## BEGIN and END Blocks In Ruby

Syntax:

```
BEGIN{
```

```
Code
```

```
. }
```

```
END{
```

```
.
```

```
. }
```

65

## Methods or Functions

Syntax:

```
def method_name
```

```
# Statement 1
```

```
# Statement 2
```

```
.
```

```
return
```

```
end
```

66

## Methods with arguments

- Single argument
- Multiple arguments
- Default arguments
  - Can be placed anywhere!
- Variable number of arguments

67

## Multiple arguments

```
def method_name(var1, var2, var3)

# Statement 1

# Statement 2

.

.

end
```

68

# Methods with variable number of arguments

Syntax:

```
def method_name(*variable_name)
# Statement 1
# Statement 2
.
.
end
```

69

## Practice Homework 1

Write a function to Reverse a string without using  
String#reverse() method

70

## Next time

- Array
- Hash
- Methods
- Iterators
- More on Range Operators
- Proc, Blocks, and Lambda
- I/O File operations
- Exceptions