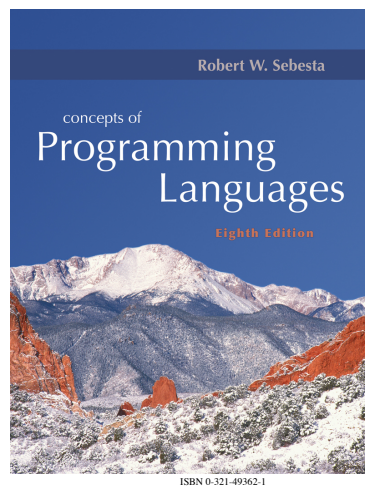


Chapter x

Ruby



Ruby

- Origins and Uses of Ruby
- Scalar Types and Their Operations
- Simple Input and Output
- Control Statements
- Fundamentals of Arrays
- Hashes
- Methods
- Classes
- Code Blocks and Iterators

2

Origins and Uses of Ruby

- Designed by Yukihiro Matsumoto; released in 1996
- Use spread rapidly in Japan
- Use is now growing in part because of its use in Rails
- A pure object-oriented purely interpreted scripting language
- Related to Perl and JavaScript, but not closely

3

Scalar Types and Their Operations

- There are three categories of data types:
 - Scalars, arrays, and hashes
- Two categories of scalars:
 - Numerics and strings
- All numeric types are descendants of the **Numeric** class
- Integers: **Fixnum** (usually 32 bits) and **Bignum**

4

Scalar Types and Their Operations (cont.)

- Scalar Literals
 - An integer literal that fits in a machine word is a **Fixnum**
 - Other integer literals are **Bignum** objects
 - Any numeric literal with an embedded decimal point or a following exponent is a **Float**
 - All string literals are **String** objects
 - Single-quoted literals cannot include characters that are specified with escape sequences
 - Double-quoted literals can include escape sequences and embedded variables *can* be interpolated

5

Scalar Types and Their Operations (cont.)

- Variables
 - Names of local variables begin with lowercase letters and are case sensitive
 - Variables embedded in double-quoted literal strings are interpolated if they appear in braces and are preceded by a pound sign (#)
 - Expressions can also be embedded in braces in double-quoted literals
 - Variables do not have types—they are not declared
 - Assignment statements assign only object addresses

6

Scalar Types and Their Operations (cont.)

- Numeric Operators
 - Arithmetic operators are like those of other languages, except Ruby has no ++ or --
 - Division of integer objects is integer division
 - Any operation on **Bignum** objects that produces a result that fits in a **Fixnum** coerces it to a **Fixnum**
 - Any operation on **Fixnum** objects that produces a result that is too large coerces it to a **Bignum**
 - The **Math** module has methods for trig and transcendental functions e.g., **Math.sin(x)**

7

Scalar Types and Their Operations (cont.)

- Interactive Ruby
 - irb is an interactive interpreter for Ruby
 - Shorten the prompt with `conf.prompt_i = ">>"`
- String Methods
 - Some can be used as infix binary operators
 - Catenation is +, which can be infix
 - Append is <<, which can be infix

8

Scalar Types and Their Operations (cont.)

- String Methods (continued)
 - Remember how assignment works!
- ```
>> mystr = "Wow!"
=> "Wow!"
>> yourstr = mystr
=> "Wow!"
>> mystr = "What?"
=> "What?"
>> yourstr
=> "Wow!"
```

9

## Scalar Types and Their Operations (cont.)

- String Methods (continued)
    - If you want to change the value of the location that is referenced by a variable, use **replace**
- ```
>> mystr.replace("Oh boy!")
=> "Oh boy!"
```
- More **String** methods:
- | | |
|------------|----------|
| capitalize | strip |
| chop | rstrip |
| chomp | rstrip |
| upcase | reverse |
| downcase | swapcase |

10

Scalar Types and Their Operations (cont.)

- String Methods (continued)
 - The *bang* or *mutator* methods - add an !
 - Strings can be indexed - `str[3]`
 - Substrings can be referenced - `str[3,5]`
 - Specific characters of a string can be assigned with []= - `str[2,3] = "gg"`
 - The `==` operator tests for equality of the objects
 - To test whether two variables reference the same object, use the `equal?` method

11

Scalar Types and Their Operations (cont.)

- String Methods (continued)
 - The `eq?` method tests whether the receiver object and the parameter have the same type and the same value
 - For ordering, use the `<=>` operator; it returns 0,1, or -1
 - The `*` operator takes a string and a numeric expression—it replicates the string

12

Simple Input and Output

- Screen Output
 - `puts` takes a string literal as its operand
 - Implicitly attaches a newline
 - Use `print` if you don't want the newline

13

Simple Input and Output

- Keyboard Input
 - The `gets` method gets a line of input from the keyboard and returns it, including the newline (returns `nil` if it gets EOF)
(UNIX: EOF is Ctrl-D; Windows: EOF is Ctrl-Z)
`name = gets`
`name = gets.chomp`
 - If the input is a number, it must be converted with either `to_i` or `to_f`
`age = gets.to_i`

14

Simple Input and Output (cont.)

- Keyboard Input (cont.)
 - If the input is a number, it must be converted with either `to_i` or `to_f`
`age = gets.to_i`
- Running a Ruby script
 - `ruby [flags] filename`
 - `-c` flag is compile only; `-w` flag is for warnings

15

Control Statements

- Control Expressions
 - `true`, `false`, `nil` (false), variables, relational expressions, or compound expressions
 - If it is a variable, it is true if its value is anything except `nil`
- Relational operators
 - usual six (like C and Java) plus `<=>`, `equal?`, and `equal?`

16

Control Statements (cont.)

- Control Expressions
 - `true`, `false`, `nil` (false), variables, relational expressions, or compound expressions
 - If it is a variable, it is true if its value is anything except `nil`
 - Relational operators
 - Usual six (like C and Java) plus `<=>`, `equal?`, and `equal?`
 - Two sets of Boolean operators:
 - Higher precedence: `&&`, `||`, `!`
 - Lower precedence: `and`, `or`, `not`
 - Operator precedence and associativity: typical

17

Control Statements (cont.)

- Selection Statements
 - `unless` is inverse of `if`
 - Multiple selection statements (see Chapter 8, pp. 353–355)
- Logical pretest loop
 - `while control_expression`
`loop body`
`end`
 - `until` has the same syntax, but opposite control

18

Control Statements (cont.)

- Unconditional Loop

```
loop
  code block
```

- The **break** statement – like other C-based languages
- The **next** statement – control goes to the first statement in the code block
- Ruby does not have a C-style **for** statement – Iterators are used instead – later

19

Fundamentals of Arrays

- Differences between Ruby arrays and those of other common languages:

- Length is dynamic
- An array can store different kinds of data

- Array Creation

- Send **new** to the **Array** class

```
list1 = Array.new(100)
All elements reference nil
list2 = Array.new(5, "Ho")
list 2 is ["Ho", "Ho", "Ho", "Ho", "Ho"]
```

20

Fundamentals of Arrays (cont.)

- Array Creation (cont.)

- Assign a list literal to a variable

```
list2 = [2, 4, 3.14159, "Fred", [] ]
```
- All array subscripts are integers and the first subscript is always zero
- The length of an array is returned by **length**

21

Fundamentals of Arrays (cont.)

- The **for-in** Statement

```
for value in list
  sum += value
end
```

- The **value** variable takes on the values of the elements of list (not references to the values)
- **irb**'s response to a **for** construct is to return an array of the values taken on by the **value** variable
- The list could be an array literal

22

Fundamentals of Arrays (cont.)

- Methods for Arrays and Lists

- Adding and deleting the end elements

```
unshift, shift, push, pop
```
- The **concat** method
 - Takes one or more parameters, which are pushed onto the end of the array
- The **reverse** method
 - Does what its name implies
- The **include?** predicate method
 - Searches the array for the given element

23

Fundamentals of Arrays (cont.)

- Methods for Arrays and Lists (cont.)

- The **sort** method
 - For arrays of a single element type
 - Works for any type elements for which it has a comparison operation
 - Returns a new array; does not affect the array object to which it is sent
- Set operations
 - &** set intersection
 - for set difference
 - |** for set union

24

Hashes

- a.k.a. Associative Arrays
- Two fundamental differences between arrays and hashes:
 - Arrays use numeric subscripts; hashes use string values
 - Elements of arrays are ordered and are stored in contiguous memory; elements of hashes are not

25

Hashes (cont.)

- Hash Creation
 1. Send new to the Hash class

```
my_hash = Hash.new
```
 2. Assign a hash literal to a variable

```
ages = ("Mike" => 14, "Mary" => 12)
```
- Element references – through subscripting

```
ages["Mary"]
```
- Element are added by assignment

```
ages["Fred"] = 9
```

26

Hashes (cont.)

- Element removal

```
ages.delete("Mike")
```
- Hash deletion

```
ages = ()
```

 or

```
ages.clear
```
- Testing for the presence of a particular element

```
ages.has_key?("Scooter")
```
- Extracting the keys or values

```
ages.keys
```

```
ages.values
```

27

Methods

- General form:

```
def method_name[(formal_parameters)]
  statement_sequence
end
```
- When a method is called from outside the class in which it is defined, it must be called through an object of that class
- When a method is called without an object reference, the default object is self
- When a method is defined outside any class, it is called without an object reference

28

Methods (cont.)

- Method names must begin with lowercase letters
- The parentheses around the formal parameters are optional
- Neither the types of the formal parameters nor that of the return type is given
- If the caller uses the returned value of the method, the call is in the place of an operand in an expression

29

Methods (cont.)

- If the caller does not use the returned value, the method is called with a standalone statement
- The **return** statement is often used to specify the return value
- If a **return** is not executed as the last statement of the method, the value returned is that of the last expression evaluated in the method

30

Methods (cont.)

- Local Variables
 - Local variables are either formal parameters or variables created in the method
 - A local variable hides a global variable with the same name
 - The names of local variables must begin with either a lowercase letter or an underscore
 - The lifetime of a local variable is from the time it is created until execution of the method is completed

31

Methods (cont.)

- Parameters
 - Formal parameters that correspond to scalar actual parameters are local references to new objects that are initialized to the values of the corresponding actual parameters
 - Actual parameters that are arrays or hashes are in effect passed by reference
 - If a method has only normal parameters, the number of actual parameters must match the number of formal parameters

32

Methods (cont.)

- Parameters (cont.)
 - The normal scalar parameters in a method can be followed by an asterisk parameter, which is a reference to an array, which accepts any number of parameters
 - Formal parameters can have default values
 - Keyword parameters can be implemented using hash literals as actual parameters

33

Methods (cont.)

- Parameters (cont.)
 - The normal scalar parameters in a method can be followed by an asterisk parameter, which is a reference to an array, which accepts any number of parameters
 - Formal parameters can have default values
 - Keyword parameters can be implemented using hash literals as actual parameters
 - If the hash literal follows all normal scalar parameters and precedes any array or block parameters, the braces can be omitted
- ```
find(age, {'first' => 'Davy', 'last' => 'Jones'})
```

34

## Methods (cont.)

---

- Parameters (cont.)
  - Symbols are created by preceding a string with a colon
  - Symbols can be used to specify the keys in hash literals

```
find(age, :first => 'Davy',
 :last => 'Jones')
```

35

## Classes

---

- General form:

```
class class_name
 ...
end
```
- Class names must begin with uppercase letters
- The names of instance variables must begin with at signs (@)
  - A class may have a single constructor, initialize
    - Constructors initialize the instance variables
    - Constructors can have parameters

36

## Classes (cont.)

---

- Classes are dynamic – subsequent definitions can include new members; methods can be removed with `remove_method` in subsequent definitions
- Access Control and other details of classes
  - see Chapter 12 (pp. 540–543)

37

## Code Blocks and Iterators

---

- A block is a segment of code, delimited by either braces or `do` and `end`
- By itself, a block does nothing
- Blocks can be sent to methods by attaching them to calls
  - This construct is used to build iterators
- Built-in Iterators
  - `times` – blocks are sent to number objects to build simple counting loops

```
5.times {puts "hey!"}
```

38