Overview

- In the set of slides that introduced Ruby objects we saw how to invoke methods for objects:
  - arrays: max, push, pop, sort, ...
  - strings: reverse, append, ...

- Today:
  - a new class (another type of list)
  - basic I/O
  - continue learning about existing methods
  - learn how to write our own methods

Reading

- first part of Chapter IV of Learning Ruby

Examples

- We can use an associative array to implement a phone book:
  - keys: names of people
  - contents: their phone numbers
    - phonebook["john"]
    - "346-3973"
    - phonebook["steve"]
    - "346-3964"

- A second example: a color map (key = color name, contents = RGB values)
  - color["clover"]
    - [0, 128, 0]
  - color["moss"]
    - [0, 128, 64]
  - color["seafoam"]
    - [0, 255, 128]
Making Associative Arrays

- Associative arrays in Ruby are defined by the class named **Hash**
- “hash table”, “hash”, and “map” are other names for associative arrays
- Two ways to make a new empty hash:
  - `a = Hash.new`
  - `a = {}`
- Note the “curly braces” instead of “square brackets”
- `{ }` is an empty associative array
- `[ ]` is an empty array

Making Associative Arrays (cont’d)

- An associative array with predefined values is a list of key-value pairs
  - use the `=>` operator
  - put the key on the left, the value on the right
- Examples:
  ```ruby
  phonebook = {"john" => "346-3973", "steve" => "346-3964"}
  color = {
    "clover"=>[0, 128, 0],
    "moss"=>[0, 128, 64],
    "seafoam"=>[0, 255, 128],
    "fern"=>[64, 128, 0]
  }
  ```
  Ruby allows you to spread expressions over several lines if it makes your code more readable

Adding New Items

- To add a new item to a hash, simply make an assignment
  ```ruby
  >> phonebook = {"john" => "346-3973", "steve" => "346-3964"}
  >> phonebook["jan"] = "346-3973"
  >> phonebook
  => {"john"=>"346-3973", "jan"=>"346-1375", "steve"=>"346-3964"}
  >> color["lime"] = [0,255,0]
  >> color
  => {"clover"=>[0, 128, 0], "moss"=>[0, 128, 64], "lime"=>[0, 255, 0], "seafoam"=>[0, 255, 128], "fern"=>[64, 128, 0]}
  ```

Array vs Hash

- Why does Ruby have both types of arrays?
  - why not make every array an associative array?
- Main difference between an array and a hash:
  - an array is an ordered collection
  - hashes are unordered
  - notice from the previous slide that newly inserted items aren’t at the end...
- There are other differences, mainly in how they are implemented
  - e.g. it is usually more efficient to retrieve an item from an array
- Note: a hash would be a good candidate for implementing the seen list in the second version of the “matching pairs” algorithm
  - key = number from the array being scanned
  - seen[a[i]] = true adds a[i] to the hash
Writing Your Own Methods

- Programs can become very long and complicated to make it easier to manage, break a complex program into smaller pieces
- An old term for a small part of a program is a “subroutine”
  - the top level program was known as the “main routine” or “main”
  - some other terms for the same basic concept: procedure, function
- A method is a subroutine that is usually attached to an object
  - but many methods “stand alone” and can be used by themselves
    - example: readlist from Lab #1
- This next set of slides: how to write a new method in Ruby

Defining a Method

- The general form of a method definition in Ruby:
  ```ruby
def name(params)
  statement
  statement...
end
```
- def stands for “define”
- The items enclosed in parentheses following the method name are parameters
  - these are items from the rest of the program that you want your method to use

Example: Insertion Sort

- To make a sort method we can use in any program, put these lines in a file named isort.rb:
  ```ruby
def isort(a)
  for j in 1..a.length-1
    key = a[j]
    i = j-1
    while i >= 0 && a[i] > key
      a[i+1] = a[i]
      i = i-1
    end
    a[i+1] = key
  end
end
```
- Note: these nested loops are from an earlier set of slides (“Ruby Loops”)

Example: Insertion Sort (cont’d)

- The name of the method we are defining
- a is a parameter
- Inside the body of the method a refers to the array passed in when isort is called
How to Use the New Method

- First read the method in from the file:
  ```ruby
  load "isort.rb"
  => true
  ```

- Now you can pass it the name of an array to sort:
  ```ruby
  nums = [2, 4, 3, 4, 1, 4, 5, 1]
  => [2, 4, 3, 4, 1, 4, 5, 1]
  isort(nums)
  => [1, 1, 2, 3, 4, 4, 4, 5]
  ```

Return Value

- What is the value returned when Ruby calls our method?
- For the sort method of the array class, the value is the sorted array:
  ```ruby
  nums = [2, 4, 3, 4, 1, 4, 5, 1]
  => [2, 4, 3, 4, 1, 4, 5, 1]
  nums.sort
  => [1, 1, 2, 3, 4, 4, 4, 5]
  ```

- What is printed when we call our own method?
  ```ruby
  isort(nums)
  => 1..7
  ```
  Hmm

Return Value (cont’d)

- To specify a return value, use the return statement:
  ```ruby
def isort(a)
  for j in 1..a.length-1
    key = a[j]
    i = j-1
    while i >= 0 && a[i] > key
      a[i+1] = a[i]
      i = i-1
    end
    a[i+1] = key
  end
  return a
end
  ```

- Other return values are true/false (indicating success or not), etc

A Note About Parameters

- Question: when we pass an array to our isort method, do we pass the array, or a copy of the array?
  ```ruby
  nums = [2, 4, 3, 4, 1, 4, 5, 1]
  isort(nums)
  => true
  ```

- What will Ruby print?
  - (a) [2, 4, 3, 4, 1, 4, 5, 1]
  - (b) [1, 1, 2, 3, 4, 4, 4, 5]

- Ans: (b)
  - Ruby passes a reference to the array, and any modifications made inside the method are reflected in the original object....