The 3Rs of Script Programming

Readin’
Rritin’
Runnin’
I/O

- In general a computer program reads data and options, carries out some calculations, and prints results.

- Input comes from:
  - files
  - command line arguments
  - user interactions (keystrokes, mouse clicks, ...)
  - laboratory instruments and other special-purpose hardware (“embedded computing”)

- Output data goes to:
  - files
  - output devices like printers and terminals
  - displays
  - special purpose hardware, e.g. controllers
I/O (cont’d)

❖ In a script, the program also interacts with other programs
  ✦ run another command line program
  ✦ pass data to the other program, which it reads as input
  ✦ read data written by the other program
  ✦ create “events” that the other program interprets as keystrokes or mouse clicks

❖ The goal for this set of slides is to explore how to do I/O in Ruby
  ✦ opening files and reading the contents
  ✦ creating new output files
  ✦ running other applications

❖ Future slides will expand on these ideas
  ✦ reading from and writing to a database
  ✦ accessing remote data from an internet service
❖ Many of the concepts described in these slides originated with (or were refined by) the Unix operating system
   ✧ started as a personal “programmer’s workbench” at AT&T in the 1970s
   ✧ grew into a general-purpose computing environment
   ✧ research projects at UC Berkeley, other places helped it spread in academia
   ✧ became a de facto standard with the introduction of workstations (Sun, Apollo, ...) in the 1980s
❖ Variants of Unix were sold along with workstations and larger computers
   ✧ Sun: Solaris
   ✧ HP: HPUX
   ✧ SGI: Irix
❖ In 1988 IEEE adopted a standard known as POSIX to provide a uniform set of requirements
   ✧ programmers could write code on one POSIX-compliant system, expect I/O to be the same on others
Linux, OS/X, and Windows

❖ Mac OS/X is fully compliant with the POSIX standards
❖ Varieties of Linux are mostly compliant
  ✦ I/O operations and most other things you want to do will probably work
  ✦ there are only minor variations from the standard
❖ There are software packages for Microsoft Windows that implement POSIX operations
  ✦ cygwin
  ✦ Microsoft Services for Unix

❖ Throughout the remainder of these slides when I refer to “Unix” I mean “POSIX”

❖ Today’s slides: POSIX concepts (files, streams, processes)
  ✦ next week: Ruby methods that interact with the host operating system, e.g File.open
  ✦ Ruby builds on the POSIX interface, allows you to operate at a more abstract level
  ✦ but for writing scripts you’ll want a pretty good understanding of POSIX commands
Shells

- When you interact with a Unix system via a command line, you are actually working with a program known as a **shell**
  - the shell prints a prompt
  - when you type a line, the shell expects the first word to be the name of a program
  - the shell asks the Unix system to run the program
  - the shell collects the remaining words on the line and puts them in ARGV for the program to use

- There are many different shell programs
  - **sh** is a very old one
  - **bash** is a newer version of **sh**
  - **csh** is an old shell with a “C-style” syntax for writing shell scripts
  - **tcsh** is a newer version of csh

- If you have experience with one or the other stick with it
  - otherwise I recommend **tcsh**
  - it has lots of nice features for terminal interaction for “power users”
Streams

- The main concept in Unix I/O is a **stream**
  - an input stream is a sequence of characters read by an application
  - an output stream is a sequence of characters written by an application

- The advantage, for programmers, is that one only needs to use methods to read and write streams
- An input stream might be the contents of a file, keystrokes typed by a user, or output from another program
- The programmer just needs to call a method to read from a stream
Characters

- In Unix streams are sequences of characters
  - originally 7-bit ASCII characters
  - now there are options for Unicode and other “wide characters”
- With 7 bits there are $2^7 = 128$ different characters
  - 96 printing characters for letters, digits, symbols
  - 32 “control” characters to mark ends of lines and other special situations
- The term “control character” goes back to the days when output was printed on a teletype or line printer
  - control characters moved the paper or print head
  - printing characters were typed on the paper
  - control characters included tab, return, line feed, form feed, bell, ...
Text Files

- Most data files in a Unix system are text files
- Think of a file as a sequence of characters
  - the last character in the file is the EOF (end of file) character
- Example: here are the first few lines of a data file used in Project 2

```
[fintan:programs] % head -3 cars.txt
allroad     1969     82
gti         2305    105
prius       5696    118
```

```
[fintan:programs] % od -t x1 cars.txt
00000000  61 6c 6c 72 6f 61 64 20 20 20 20 20 31 39 36 39
00000200  20 20 20 20 20 20 20 20 20 20 20 20 38 32 0a
00000400  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00000600  70 72 69 75 73 20 20 20 20 20 20 20 20 20 20
00001000  20 20 20 20 20 20 20 20 20 20 31 31 38 0a
33 33 30 69 20 20 20 20 9
```

look up “ASCII” on Wikipedia to see what these codes represent

- `% head -n file` print the first n lines in file
- `% od file` “octal dump” -- print the characters in file
Standard Streams

- Every Unix application has three standard streams
  - `stdin` the default source of input text
  - `stdout` the default destination for output text
  - `stderr` a stream for printing error messages
- We’ve been using two of these streams in our Ruby programs
  - `gets` reads a string from stdin
  - `puts` prints a string on stdout
- In Ruby `gets` and `puts` work on “lines” of text
  - `gets` reads the next line from stdin
  - a line is defined to be the sequence of characters up to and including the next newline
  - `puts` prints a string and a newline
- The idea of a “line” is not part of the POSIX standard
  - it’s a convention implemented in Ruby and some other languages
Streams and the Command Line

- When you run a program from the command line the shell connects stdin and stdout to data sources
  - the default for stdin is your keyboard: every key you type is sent to the application
  - the default for stdout is your terminal window: every character sent to stdout is displayed on your screen

- Example: the line length program (ll.rb) reads lines and then prints them back out, along with the number of characters on the line

```ruby
#!/usr/bin/env ruby

while line = gets
  puts line.length.to_s + "": " + line
end
```
Streams and the Command Line (cont’d)

- If we run this program with no additional command line arguments:

```
[fintan:programs] % ll.rb
hello, world
13: hello, world
```
```
ciao
5: ciao
```

- Why did the program say there were 5 letters in “ciao”? And 13 in “hello, world”? 
  - because it counted the newline character that ended each input line

- **Note:** saying “every key you type is included” in the line sent to the program is not exactly true
  - the system lets you edit the line, e.g. hit backspace to correct errors
  - these special editing keystrokes are not put in the input stream

- **Note:** type ^D (control-D) to terminate the input stream
Redirection

- Unix shells allow you to **redirect** the standard streams
- Use the `<` character on the command line to connect a file to a program’s stdin stream
- Example: to count the number of characters on each line in cars.txt:

  ```
  [fintan:programs] % ll.rb < cars.txt
  24: allroad 1969 82
  24: gti 2305 105
  24: prius 5696 118
  ```

- Use the `>` character to redirect stdout to a file, e.g.

  ```
  [fintan:programs] % ll.rb > temp.txt
  hello, world
  ciao
  ```

  ```
  % cat file
  “catenate” -- copy the contents of file to stdout
  ```
Redirection (cont’d)

❖ Be careful when you use output redirection with >
   ✦ if the output file exists, it will be erased and over-written with the output from the program
   ✦ no warnings*, no trash with the old version, no backup -- it’s gone
   ✦ * there is a way to set up your shell to prevent this from happening accidentally

❖ If you use a double >> the output from the program is appended to the file
   ✦ this command will pass the contents of books.txt to the ll.rb program, and append the output to temp.txt:

   [fintan:programs] % ll.rb < books.txt >> temp.txt
Additional Streams

- Programs can open additional streams
- In Ruby, a File object represents a connection to a file
  - calling the open method creates a stream and connects it to a specified file
  - after creating a File object, use the object's gets method to read a line from the file

```ruby
>> f = File.open("cars.txt")
=> #<File:cars.txt>
>> f.gets
=> "allroad     1969     82\n"
>> f.gets
=> "gti         2305    105\n"
```
Programs that open additional streams typically expect the user to supply the name of the file as a command line parameter

% cracker.rb wordlist.txt /etc/password
or
% cracker.rb wordlist.txt < /etc/password

In this case, it is up to the programmer to get the filename from ARGV and pass it as a parameter in a call to File.open
File Names on the Command Line

- Recall this little program from the last set of slides -- it prints out the contents of ARGV
- I added three lines at the bottom to print all the lines in stdin

```ruby
#!/usr/bin/env ruby
# What is ARGV?

puts "ARGV is a #{ARGV.class}"
puts "it has #{ARGV.length} elements"
ARGV.each_with_index do |x, i|
  puts "ARGV[#{i}] = #{ARGV[i]} (a #{ARGV[i].class})"
end
while line = gets
  puts "'#{line.chomp}'"
end
```

The `chomp` method cuts off the last character in a string if it is a newline
File Names on the Command Line

❖ Here’s what happens when we run the program with no command line arguments:

```
[fugu:programs] % args.rb
ARGV is a Array
it has 0 elements
hello from the terminal
'hello from the terminal'
```

❖ Exactly what I expected:

✦ the program said there was nothing in ARGV
✦ since it was expecting something on stdin it waited for me to type something
✦ every line I typed was echoed, surrounded by single quotes

```
while line = gets
  puts "'#{line.chomp}'"
end
```
Now let’s run it again, using redirection to get input lines from a file

[fugu:programs] % args.rb < cars.txt

ARGV is a Array
it has 0 elements

'allroad     1969     82'
'gti         2305    105'
'prius       5696    118'

Note that again, as expected, Ruby said there were 0 items in ARGV
❖ the Unix shell processes the redirection character
❖ there is nothing left on the command line when our program starts
❖ Another note: the program exits the while loop after it reads the last line
Here's where things get a bit confusing.

What happens if we run the program with file names as command line arguments?

```
[fugu:programs] % args.rb cars.txt books.txt
ARGV is a Array
it has 2 elements
ARGV[0] = cars.txt (a String)
ARGV[1] = books.txt (a String)
'allroad     1969     82'
'gti         2305    105'
'prius       5696    118'
'latex         1    23.99'
'mysql         0    29.99'
'dreamweaver   0    59.45'
```

Ruby opens up each file, and connects it to stdin.

```
while line = gets
  puts "'#{line.chomp}'"
end
```

when you get to the end of one file, it opens the next one.
Last test: what happens if you supply an argument that is not a file name?

[fugu:programs] % args.rb foo.txt
ARGV is a Array
it has 1 elements
ARGV[0] = foo.txt (a String)
args.rb:16:in `gets': No such file or directory - foo.txt
(Errno::ENOENT)
   from args.rb:16

Summary: if your program calls gets
✦ if there are no arguments in ARGV, Ruby will use the default stdin
✦ (meaning the stream supplied by Unix, which is either the terminal or a redirected file)
✦ if there are strings in ARGV, Ruby automatically passes them to File.open, and connects each one to stdin
Pipes

- There are many situations where a user wants to run a program to create some data, then pass that data to a second program.

- Example: suppose I want to know how many Ruby programs are in my directory.

```
[fugu:programs] % ls *.rb > listing.txt
[fugu:programs] % more listing.txt
alias.rb*
args.rb*
bmi.rb*
choose.rb*
...
[fugu:programs] % wc listing.txt
  36 36 386 listing.txt
```

- `% ls` list the files in the current directory.
- `% more file` print the contents of file, one screenful at a time.
- `% wc file` “word count” -- print the number of line, words, chars.
Pipes

- Unix shells provide a short-cut
- Use the “pipe symbol” to tell the system to start two programs
  - the standard output stream of the first program is connected to the standard input stream of the second

  ```bash
  [fugu:programs] % ls *.rb | wc
  36      36     386
  ```

- This is a very useful feature
  - without it you would always be littering your working directory with temporary files
  - need to think of a name, worry about whether it’s useful data, figure out when it can be deleted, ....
  - if the first program is going to run for a while, you don’t have to wait around for it to finish
  - your system will start the program, but it will wait until data arrives via the pipe
POSIX I/O in Ruby

- The previous slides introduced the main concepts of I/O in POSIX systems
  - applications typically read and write *streams* of data
  - an application has three predefined streams (*stdin*, *stdout*, *stderr*)
  - the *shell* connects the application’s streams to data sources
    - default: terminal
  - use *redirection* to connect a stream to a file
    % foo < input.txt
    % foo > output.txt
  - an application can create additional streams, e.g. open a file named on the command line
  - use a *pipe* to connect the output of one program to the input of another
    % foo < input.txt | bar > output.txt

- The next set of slides will show how to make use of these concepts in Ruby programs
Kernel Methods

- You have already seen many examples that use kernel methods for I/O
  - `gets` reads the next line from `stdin`
  - `puts` prints a string to `stdout`

- Here are some more kernel methods to know about (examples on the next slides)
  - `readlines` create an array containing each line from `stdin`
  - `p` print an object (calling its `inspect` method)
  - `print` print one or more objects
  - `getc` get the next character from `stdin`
  - `putc` print a character to `stdout`
The `readlines` method reads all the lines in `stdin` and returns them in an array:

```ruby
>> a = readlines
goodbye
cruel
world
^D
=> ["goodbye\n", "cruel\n", "world\n"]
>> a[2]
=> "world\n"
```

Why would you use this method?
- Sometimes it’s easier to use `Array` methods to work on lines from the input stream.
- E.g. look for a line with a certain pattern, then get something from the previous line.

But beware that making a huge array in memory for all the lines in a very large file might not be the best approach.
Kernel Methods

- `puts`, `print`, and `p` are three different ways to print something to `stdout`
  - `puts` takes one or more objects as parameters
  - it calls an object’s `to_s` method to make a string, then prints each string, one per line

```ruby
>> puts "hello"
hello
=> nil
>> puts 3
3
=> nil
>> puts 3, "little", "pigs"
3
little
pigs
=> nil
```

Note the return value from a call to `puts` is always `nil`
Kernel Methods

- `print` is like `puts`, but it prints the items one after another on a single line.
- Like `puts`, it calls each object’s `to_s` method to make a string.

```ruby
>> x = 4
=> 4
>> print "x = ", 4, "\n"
x = 4
>> print [3, "little", "pigs"]
3littlepigs
```

- As this last example shows, when an object is not a number or a string `to_s` probably doesn’t do what you want.

```ruby
>> h = {"one" => 1, "two" => 2}
=> {"two"=>2, "one"=>1}
>> print h
two2one1
```
Kernel Methods

- p is the same as print except it calls a method named inspect instead of to_s

```
>> p [3, "little", "pigs"]
[3, "little", "pigs"]
=> nil
>> p h
{"two"=>2, "one"=>1}
=> nil
```

- Every object has an inspect method
  - it is intended to print a “readable” description of the object

- Note that irb uses p to print the value of an expression
  - if you want your output to match what you see in irb use p
Characters

- Ruby does not have a Character class
- Individual items from String objects and I/O streams are integers
  - these integers are ASCII codes for the items in the string or stream

```ruby
>> s = "hello"
=> "hello"
>> s[0]
=> 104

>> i = getc
hello
=> 104
>> getc
=> 101
```
To make it easier to write programs that deal with individual characters, Ruby has methods to convert back and forth between ASCII codes and letters.

- To make a one-letter string from an ASCII code, use the `chr` method of the `Fixnum` class.
  ```ruby
  n = 104
  => 104
  n.chr
  => "h"
  ```

- To create an integer with the ASCII code for a character, put a `?` in front of it.
  ```ruby
  s = "hello"
  => "hello"
  s[0]
  => 104
  s[0] = ?j
  => 106
  s
  => "jello"
  ```
Streams

- Streams in Ruby are instances of the IO class
- This class has `gets`, `puts`, and other kernel methods, plus many more
- When your Ruby program starts, the system automatically makes three IO objects for you, one for each of the default streams
  - STDIN
  - STDOUT
  - STDERR
- If you want to do a simple operation like `puts`, you can either call the kernel method or the IO object’s method
- Why would you want to use an IO object instead of a kernel method?
  - the IO class has many useful methods not available in the Kernel module
  - example: the `lineno` (“line number”) method tells you how many lines you have read from an input stream
  - IO also has a method named `read`: `STDIN.read` reads the entire input stream and puts it in a String object
Stream Iterators

- Another good reason to use an IO object: the IO class has iterators
- Iterators for containers let us “traverse” the container to access each element

```ruby
>> a = ["latex","sqlite3","ls","mysql"]
=> ["latex", "sqlite3", "ls", "mysql"]
>> a.each {|x| puts x.length }
5
7
2
5
```

- The `each` iterator in the IO class invokes the block for each line in the stream, e.g.

```ruby
>> STDIN.each {|x| puts x.chomp.length}
hello
5
world
5
```
Stream Iterators

- IO also has an iterator named `each_byte`

```ruby
>> STDIN.each_byte { |ch| puts ch }
hello
104
101
108
108
108
111
10
```
Example: ls.rb

This program prints a “listing” of a program it reads from stdin

```ruby
#!/usr/bin/env ruby -wKU
STDIN.each do |line|
  printf "%4d:  %s\n", STDIN.lineno, line.chomp
end
```

Example:

```
% ls.rb < args.rb
  1:  #! /usr/bin/env ruby
  2: 
  3:  # What is ARGV?
  4: 
  5:  puts "ARGV is a #{ARGV.class}"
  6:  puts "it has #{ARGV.length} elements"
```
IO Streams vs Kernel

- A problem with using `STDIN.gets` is that it always reads from stdin
- If your program takes advantage of the “feature” that Ruby opens files named on the command line this call to `gets` will not read a line from the file
- Try it -- modify `args.rb` so the call to `gets` is `STDIN.gets` and see what the difference is
- Invoke both versions of the program this way:
  ```bash
  % args.rb cars.txt
  ```
- What happens in each case?

```ruby
while line = gets
  puts "'#{line.chomp}'"
end
```

```ruby
while line = STDIN.gets
  puts "'#{line.chomp}'"
end
```
Files

❖ Use a class named File when you want to do something with a file on your disk:
  ✦ open an existing file to read the contents
  ✦ create a new file to save the output of a program
  ✦ get basic information about the file itself, e.g. the date it was created

❖ The standard approach to reading from or writing to files is
  ✦ open the file by creating a new File object that has the necessary information
  ✦ use the file, reading or writing characters
  ✦ close the file, making sure all buffers are flushed, and deallocate the object

❖ To create a new File object, you can call File.new

  
  f = File.new("cars.txt")

❖ To close it, call the object’s close method:

  
  f.close
Files

❖ A more idiomatic approach in Ruby use an alternative constructor named `open` and a block

```ruby
File.open(name) do |f|
  ...
end
```

✦ File.open calls `File.new` to make a new `File` object
✦ creates a temporary variable named `f`, and assigns the `File` object to `f`
✦ executes the code in the body of the block
✦ closes the file

❖ Example:

```ruby
>> File.open("cars.txt") { |f| puts "created \#{f.ctime}" }
created Sun Feb 03 11:24:52 -0800 2008
```
File Methods

- The File class has `gets`, `puts`, and other methods we saw earlier for streams (IO objects)
- It also has `each` and `each_byte` iterators
- A typical way to write a loop that opens a file, reads each line, and does something to that line is
  ```ruby
  File.open(name).each do |s|
    ...
  end
  ```
- In this example
  - `File.open(name)` creates a new File object
  - we then invoke this object’s `each` method, which is an iterator
  - for each line in the file the iterator reads the line, stores it in `s`, and executes the block of code
The reason we can call `puts`, `gets`, and other methods of the `IO` class for objects of the `File` class is that **File objects are IO objects**

- in the terminology of object-oriented programming, `File` is a **subclass** of `IO`
- we also say `File` **inherits** the attributes of `IO`

The people who wrote the code for the `File` class defined it this way:

```ruby
class File < IO
  ...
end
```

This tells Ruby that every method defined for an `IO` object should also apply to `File` objects.

We’ll explore these ideas of subclasses and inheritance in more detail later in the term...
Writing to Files

- When you call `File.new` or `File.open` the default is to open an existing file for reading:
  - after you make an object named `f`, you can call `f.gets`, `f.readlines`, and other methods that get information from the file.

- If you call `f.puts` Ruby will complain:
  ```ruby
  >>> f = File.open("cars.txt")
  => #<File:cars.txt>
  >>> f.puts("hummer     203     57
  ```
  `IOError: not opened for writing`
Writing to Files

❖ To create a new file to write to, pass “w” as a second argument to the method that makes the File object (new or open)
❖ Example: save “hello, world” in a file:

```
>> File.open("greeting.txt","w") { |f| f.puts "Hello, world!" }
```

✦ the call to File.open tells the OS to create a new file on disk
✦ the File object made by the call to open is stored in f
✦ the call to f.puts writes a string to the file
✦ when the block terminates the file is closed, and the OS makes sure it is written out to the disk

❖ It is also possible to open a file for reading and writing, or to open a file so you can append new lines to the end of an existing file
Review: Streams and Files in Ruby

- POSIX streams are implemented in Ruby by the `IO` class
- Ruby provides three `IO` objects for the default streams (`STDIN`, `STDOUT`, `STDERR`)
  - you can use `gets`, `puts`, and other methods to read or write one string at a time
  - for most of these operations there is a corresponding kernel method
    ```ruby
    STDOUT.puts "Hello"
    puts "Hello"
    ```
- The `IO` class also has iterators (`each` and `each_byte`) for processing entire files
- To create additional streams for accessing files by name Ruby has a `File` class
  - create a `File` object with `File.new`, call `IO` methods (`gets`, `puts`, etc), close the file
  - use the `open` method and a block to combine these three phases into a single piece of code
    ```ruby
    File.open("cars.txt").each { |s| puts "#{s.chomp}" }
    ```
- By default files are opened for reading; to create a new file pass "w" to `new` or `open`
  ```ruby
  File.open("greeting.txt","w") { |f| f.puts "Hello, world!" }
  ```
A Demo Program

- To demonstrate the various techniques for reading from a file I wrote a small “demo” program
  - download `io.rb` from the class web site

- Methods in the program demonstrate different ways to read data from a file

```ruby
def Test.lines(name)
  File.open(name) do |f|
    a = f.readlines
    puts "got #{a.class} with #{a.length} items"
    p a
  end
end
```
A Demo Program

- One way to try out the methods is to load the program into irb and then type an expression that calls a method

```ruby
>> load "io.rb"
=> true

>> Test.lines("cars.txt")
got Array with 7 items
["allroad     1969     82
", "gti         2305    105
", ... ]
=> nil

>> Test.string("cars.txt")
got String with 168 items
"allroad     1969     82\ngti         2305    105\n ..."
=> nil
```
A Demo Program

- You can also run the program from the command line
- Command line arguments are the name of the method to call and the file to test

```
[fintan:programs] % io.rb lines cars.txt
    got Array with 7 items
    ["allroad  1969   82\n", "gti      2305  105\n", "prius    5696  118\n", "330i     7233  249\n", "530i     2026  75\n", "subaru   1301  52\n", "explorer 1717  78\n"]

[fintan:programs] % io.rb string cars.txt
    got String with 168 items
"allroad  1969   82\ngti      2305  105\nprius    5696  118\n330i     7233  249\n530i     2026  75\nsubaru   1301  52\nextplorer 1717  78\n"
A Demo Program

- A “dual-purpose” program like this has a method named `main` and only one statement in the body of the program:
  ```ruby
  def main
    ...
  end

  if ! defined? IRB
    main
  end
  ```

- When the program is loaded into irb all the methods are defined, but nothing happens -- there is no top level program to run.
- When the program is called from the command line the constant IRB is not defined, so the system calls the method named `main`
A Demo Program

- A straightforward way to write `main` would be to use a `case` statement, e.g.:

  ```ruby
  def main
    method = ARGV.shift or abort "Usage: io.rb method [filename]"
    filename = ARGV.shift
    case method
    when "file" : Test.file(filename)
    when "lines" : Test.lines(filename)
    when "string" : Test.string(filename)
    else
      puts "#{\$0}: unknown test: #{method}"
    end
  end
  ```

- This code is boring and error-prone
  - every time I add a new demo I have to write the method and add a line to this case statement
  - it’s easy to type the wrong name after “when” or in the call to the new method
Ruby is a Dynamic Language

- A much cooler way to write this code takes advantage of the fact that Ruby is a dynamic language.
- I put all of the methods that demonstrate a technique for reading data into a class named `Test`:

```ruby
class Test
  def Test.iterator(name)
    File.open(name).each do |line|
      puts line.dump
    end
  end
  def Test.file(name)
    ...
  end
end
```

We’ll explore techniques for defining classes later this term.
Ruby is a Dynamic Language

- We can take advantage of a special method named `send` that is built into every class.
- For a class named `Foo` there are two ways to call a method named `x`:
  - `Foo.x`
  - `Foo.send("x")`
- The second approach takes advantage of the dynamic nature of the language, since it figures out at runtime what method to call.
- For this demo program all we have to do is get the method name from the command line, and then ask the `Test` class to execute a method of that name:

```ruby
def main
  method = ARGV.shift or abort "Usage: io.rb method [filename]"
  filename = ARGV.shift
  Test.send(method, filename)
end
```
Ruby is a Dynamic Language

- What you will see in the program is a better version that first tests to make sure the class has a method with the requested name:

  ```ruby
  def main
    method = ARGV.shift or abort "Usage: io.rb method [filename]"
    filename = ARGV.shift
    if Test.public_methods.include?(method)
      Test.send(method, filename)
    else
      puts "#{ARGV[0]}: unknown test: #{method}"
    end
  end
  ```

- This dynamic approach is much nicer
  - the program is shorter
  - no need to update the case statement every time we add, delete, or rename a method
Running Other Programs

- The next set of slides will show how to run other programs from within a Ruby program
- Here are some Unix commands we’ll use in examples

```bash
[fintan:conery] % whoami
conery

[fintan:conery] % pwd
/Users/conery

[fintan:conery] % date
Sun Feb 17 17:36:11 PST 2008

[fintan:conery] % hostname
fintan.local

[fintan:conery] % ls
Applications/  Desktop/       Library/       Personal/       Sites/
Archives/     Documents/      Movies/        Pictures/       Students/
Classes/      Downloads/       Music/         Public/         Subversion/
Department/   Faxes/          Notebooks/     Research/       SysAdmin/
```
Running Other Programs

❖ One way to run another program is to use “backticks”
❖ If you make a string using the single open quote delimiters, Ruby treats the string as a shell command
   ✦ this character is often in the upper left of a keyboard
   ✦ don’t confuse it with the apostrophe or “single close quote” usually on the lower right

❖ Note the difference between `pwd` and 'pwd'
Running Other Programs

❖ When Ruby sees a string enclosed in backticks it runs the command, and the output produced by the command is captured and put in a String object

```ruby
>> who = `whoami`
=> "conery\n"

>> what = $0
=> "irb"

>> where = `hostname`
=> "fintan.local\n"

>> puts "#{who} ran #{what} on #{where} at #{`date`}"
conery
  ran irb on fintan.local
  at Sun Feb 17 17:35:05 PST 2008
```

❖ This last example emphasizes that the string produced by a shell command usually includes a newline

❖ a common idiom is `s = `command`.chomp`
Aside: Quotes

- We’ve now seen three different types of quotes in Ruby
- All three create a String object

```ruby
>> s = "Hello, World"
=> "Hello, World"

>> t = 'Hello, World'
=> "Hello, World"

>> s == t
=> true

>> u = `Hello, World`
( ) => ""

>> u = `hello.rb`
=> "Hello, World\n"

>> t == u
=> false
```
Quotes

- Strings made with double quotes are very flexible
  - include special characters, e.g. \n for newline, \t for tab
  - use “variable interpolation” with #{x}

  ```ruby
  >> s = "Hi\tWorld"
  => "Hi\tWorld"
  >> puts s
     Hi     World
  => nil
  >> greeting = "Aloha"
  => "Aloha"
  >> s = "#{greeting}, World"
  => "Aloha, World"
  ```

- But strings made with single quotes use all characters exactly as they appear

  ```ruby
  >> t = '#{greeting}, World'
  => "\#{greeting}, World"
  ```
The rules for command strings with backquotes are the same as for double quotes

```
>> args = ""
=> ""

>> a = `ls #{args}`.split
=> ["10words.txt", "alias.rb", "args.rb", ... "wordlist.txt"]

>> args = " -t"
=> " -t"

>> a = `ls #{args}`.split
=> ["hello.rb", "ex.aux", "ex.log", ... ]

>> `hostname

whoami`
=> "fintan.local\nconery\n"
Running Other Programs

❖ A second way to run another program is to use a kernel method named `system`

```ruby
>> res = system("ls")
10words.txt  efla.txt  gref.rb  overlap.rb
alias.rb     evo.rb    grep.rb  overload.rb
...
diffs.rb     gpa.rb    mpga.rb  wordlist.txt
efla.fa      greeting.txt  myhouse.txt
=> true

>> res = system("foo")
=> false
```

❖ Note the result from a call to `system` is a boolean -- `true` means the command was successful and `false` means it failed for some reason
system(x) vs `x`

- Why are there two ways to run another program?
  - the difference is in how the stdout stream of the other program is handled
  - with system(x) Ruby runs x, and Ruby’s own stdout stream is used by the other program
  - with `x` Ruby runs the other program, and everything the other program writes is captured and returned to Ruby

- This program (echo.rb) illustrates the difference:
  ```ruby
  s = gets
  res = system("echo #{s}")
  puts "res = '#{res}'"
  ```
system(x) vs `x`

- If we run the program without redirection input and output uses the terminal window

  ![Example Output]

- If we redirect the output, everything printed by echo.rb -- including things printed by the call to system -- goes to the file

  ![Example Output]

```ruby
s = gets
res = system("echo #{s}")
puts "res = '#{res}'"
```
Running Other Programs

- There are many other ways to interact with other programs
  - call `IO.popen` to open a Unix pipe
  - strings you write to the pipe are read by the other program on its stdin
  - things printed to stdout by the other program can be read from the pipe

- There are Ruby methods that invoke low level Unix functions (`fork`, `exec`, etc) that start other programs to run in parallel with your application

- Read about these in the Thomas book...
Interacting with the File System

❖ Scripts often need to interact with the file system
  ✦ make sure a file exists
  ✦ copy a file
  ✦ make a new directory
  ✦ change to another directory
  ✦ list the contents of all or part of a directory
  ✦ get attributes of a file or directory (who owns it, when it was created, ...)
  ✦ set attributes of a file or directory

❖ With Ruby programs, you can do these operations with Unix shell commands
  ✦ e.g. `system("mkdir tmpdir")`

❖ A better idea is to use methods in Ruby’s `File` and `Dir` classes
  ✦ implements operations defined for POSIX systems
  ✦ more likely to work on any POSIX compliant system
File

- A few of the useful methods in the File class (see Thomas for details, and for other methods):

  `File.basename(s)`
  `File.dirname(s)`

  - return the directory name or file name of a string, e.g. in Unix

    ```ruby
    >> p = "/Users/conery/Classes/199/programs/grep.rb"
    => "/Users/conery/Classes/199/programs/grep.rb"
    
    >> File.basename(p)
    => "grep.rb"
    
    >> File.dirname(p)
    => "/Users/conery/Classes/199/programs"
    ```

  - note that other systems have a different character in a path name (e.g. `\`?)
Test attributes of a file:

File.exists?(name)
File.readable?(name)
File.directory?(name)
File.size(name)

>> File.exists?("wordlist.txt")
=> true
>> File.readable?("wordlist.txt")
=> true
>> File.directory?("wordlist.txt")
=> false
>> File.file?("wordlist.txt")
=> true
>> File.size("wordlist.txt")
=> 585858
File

❖ Change attributes of a file:

**File.chmod(mode,name)**

✦ the first argument is a number the system translates into a set of access modes (see next slide for details)

✦ some common numbers to use:

0444  read only for all users

0644  readable and writable by the owner, read only for others

0755  executable for all, writable by the owner

>> File.chmod(0644,"wordlist.txt")

=> 1

❖ Delete or rename a file:

**File.delete(name)**

**File.rename(oldname,newname)**
Hexadecimal, Octal, Binary

- Numbers in a machine’s memory (RAM or disk) are transmitted over communication channels are binary numbers
  - it’s easier and more efficient to manufacture devices that have only two discrete states
  - voltage levels, frequencies, magnetic flux, light levels, ...

- When you type a number in irb or put a numeric constant in your program, Ruby translates the series of digits into binary
  - it assumes you intend the digits to carry their normal weight, i.e. you entered a decimal number
    
    $\gg x = 15$
    $=> 15$
    $\gg x * 2$
    $=> 30$

    - by default the string "15" means “1 times 10 plus 5 times 1"
Hexadecimal, Octal, Binary

- You can ask Ruby to interpret the digits in a different number system.
- Precede a number by a short string to identify the base:
  - `0b`     binary
  - `0`      octal
  - `0x`     hexadecimal (base 16)

- Examples:
  - `>> x = 0x15`  \(0x15\) means “1 times 16 plus 5”
    - `=> 21`
  - `>> 2 * x`  \(0777\) means “7 times 64 plus 7 times 8 plus 7”
    - `=> 42`
  - `>> x = 0777`  \(0777\) means “7 times 64 plus 7 times 8 plus 7”
    - `=> 511`
The previous examples showed that Ruby prints numbers in decimal.

To print a number in a different base use `sprintf` (or `printf`) to make a string of digits in that base.

```ruby
>> n = 226
=> 226
>> sprintf "%x", n
=> "e2"
>> sprintf "%X", n
=> "E2"
>> sprintf "%o", n
=> "342"
>> sprintf "%b", n
=> "11100010"
```
Hexadecimal, Octal, Binary

- File access modes (parameters to File.chmod) are numbers that decode into permissions.
- A mode is a 9-digit binary number:
  - first 3 bits: permissions for the file’s owner
  - second 3 bits: permissions for members of the owner’s group
  - last 3 bits: permissions for all other users
- The three bits in a group represent, in order:
  - read access
  - write access
  - execute access
- Example:
  - mode 110100000 means “the owner can read and write the file, group members can read the file, others cannot access the file”
Hexadecimal, Octal, Binary

- When you use the `-l` switch with the `ls` command Unix shows you a file’s permissions.
- Example:
  ```
  % ls -l p*
  -rw-r--r--@ 1 conery conery  441 Feb  4 14:45 password
  -rwxr-xr-x@ 1 conery conery  501 Feb  4 19:44 perimeter.rb*
  ```
- The mode bits are shown as three groups of letters “rwx”
- If an access bit is set the mode letter is shown, otherwise the mode is shown as –
  ```
  0644 = 0b110100100 = rw-r--r--
  ```
Dir

- Ruby's `Dir` class represents a directory

  **Dir.getwd()**
  - returns a string containing the full path name of the current working directory

  **Dir.chdir(p)**
  - change the current working directory to `p`

  **Dir.mkdir(p)**
  - create a new directory with name `p`

```
>> Dir.getwd()
=> "/Users/conery/Classes/199/programs"
>> Dir.mkdir("dtest")
=> 0
>> Dir.chdir("dtest")
=> 0
>> File.open("data.txt","w") { |f| f.puts "this is dtest/data.txt" }
=> nil
>> Dir.getwd()
=> "/Users/conery/Classes/199/programs/dtest"
```
Aside: Unix Paths

❖ A string that represents a file name is often called a **path**

❖ From a previous example:

    >> p = "/Users/conery/Classes/199/programs/grep.rb"
    => "/Users/conery/Classes/199/programs/grep.rb"

    >> File.basename(p)
    => "grep.rb"

    >> File.dirname(p)
    => "/Users/conery/Classes/199/programs"

❖ If a path starts with / it is an **absolute path** from the root of the file system

❖ A path starting with a word is a **relative path**
   ✦ relative paths start from the current working directory
Aside: Unix Paths

❖ Path examples:

  >> Dir.getwd
  => "/Users/conery/Classes/199/programs"
  >> Dir.mkdir("/Users/conery/Classes/199/programs/test1")
  => 0
  >> Dir.mkdir("test2")
  => 0
  >> system("ls -ld test*")
  -rw-r--r-- 1 conery conery 19 Feb 17 19:21 test.txt
  drwxr-xr-x 2 conery conery 68 Feb 18 08:25 test1
  drwxr-xr-x 2 conery conery 68 Feb 18 08:25 test2

❖ The same conventions (absolute path starts with /, relative path starts with a word) are used in chdir and other File and Dir methods that take pathname parameters
Aside: Unix Paths

❖ Pathnames also have the following shorthand notations

- the current directory ./programs
- the parent of the current directory ../www/projects

```ruby
>> Dir.chdir("/Users/conery/Classes/199/programs")
=> 0
>> Dir.chdir("..")
=> 0
>> Dir.getwd
=> "/Users/conery/Classes/199"
```

❖ Shells and editors (but not Ruby) also recognize these abbreviations:

- your home directory ~/Classes
- User fred’s home directory ~fred/projects
Aside: Unix Paths

Pathnames can also be built using pattern matching characters

* matches 0 or more letters or digits
  *.txt, *.rb
?
  matches exactly one letter or digit
  project?

```bash
>> system("ls -d *")
10words.txt  ex.tex  mpga.rb
alias.rb  f2c.rb  myhouse.txt
args.rb  factorial.rb  overlap.rb
...
```

```bash
>> system("ls *.rb")
alias.rb  factorial.rb  loop.rb
args.rb  file.rb  ls.rb
bmi.rb  foo.rb  makeapp.rb
...
```
Dir

**Dir.entries(p)**

**Dir.glob(p)**

- make an array of names of items in a directory; Dir.entries returns all items, Dir.glob returns items that match a pattern

  >> Dir.entries("/Users/conery/Classes/199/programs")
  => [ ".", ".", "10words.txt", "alias.rb", "args.rb", ..... ]

  >> Dir.getwd
  => "/Users/conery/Classes/199/programs"

  >> Dir.glob("*").length
  => 63

  >> Dir.glob("*.rb").length
  => 42

**Dir.delete(p)**

- delete a directory named p
Case Study: notebooks.rb

- I use an application named Scrivener to manage my notebooks
- Notebooks are directories with names that end in .scriv
- I keep all my notebooks in a top level directory named Notebooks

```
[fugu:conery] % cd Notebooks
[fugu:Notebooks] % ls -d *.scriv
Classes.scriv/ Introns.scriv/ Sporobolomyces.scriv/
Conferences.scriv/ Microbes.scriv/ Students.scriv/
Department.scriv/ PIP.scriv/ SysAdmin.scriv/
```

- A problem: I use more than one machine
  - if I update a notebook on my desktop (fugu) but then continue working on a project at home on my laptop (fintan) I need the latest copy
  - it’s easy to get confused about where the latest copy is
Case Study: notebooks.rb

❖ My solution:
   ✦ make sure both machines keep notebooks in the same relative location (~/.Notebooks)
   ✦ one machine will have the current version, the other(s) will have backups
   ✦ a script on fintan will check out a notebook -- copy the current version from fugu, mark the
     version on fugu as checked out
   ✦ another script will do the opposite -- copy a notebook back to fugu, mark it as checked in on
     fintan

❖ Notebooks on fugu before and after checking out Classes.scriv:
Case Study: notebooks.rb

- A script named `nco` (notebook checkout) will compress a notebook on fugu, move the compressed copy to Checked Out on fugu, copy it to fintan, and uncompress it

```shell
[fugu:conery] % nco Classes

ssh fugu.cs.uoregon.edu 'cd /Users/conery/Notebooks;
    tar zcf Classes.scriv.tgz Classes.scriv'

scp -r 'fugu.cs.uoregon.edu:/Users/conery/Notebooks/Classes.scriv.tgz' .

tar zxf Classes.scriv.tgz

rn Classes.scriv.tgz

ssh fugu.cs.uoregon.edu 'cd /Users/conery/Notebooks;
    mv Classes.scriv.tgz /Users/conery/Notebooks/Checked\ Out'

ssh fugu.cs.uoregon.edu 'cd /Users/conery/Notebooks;
    rm -rf Classes.scriv'
```
# Execute a command on the server

def remote(cmnd)
    rcmd = "cd #{NotebookDirectory}; #{cmnd}"
    puts "ssh #{Server} '#{rcmd}'"
    system("ssh #{Server} '#{rcmd}'") or abort("remote command failed: #{cmnd}")
end

# Execute a command on this machine

def local(cmnd)
    puts cmnd
    system("#{cmnd}") or abort("local command failed: #{cmnd}")
end
def checkin(filename)
    tarfile = filename + ".tgz"
    local("tar zcf #{tarfile} #{filename}")
    local("scp #{tarfile} #{Server}:#{NotebookDirectory}")
    remote("tar zxf #{tarfile}")
    remote("rm #{tarfile}")
    local("mv #{tarfile} #{LocalCache}")
    local("rm -rf #{filename}")
end