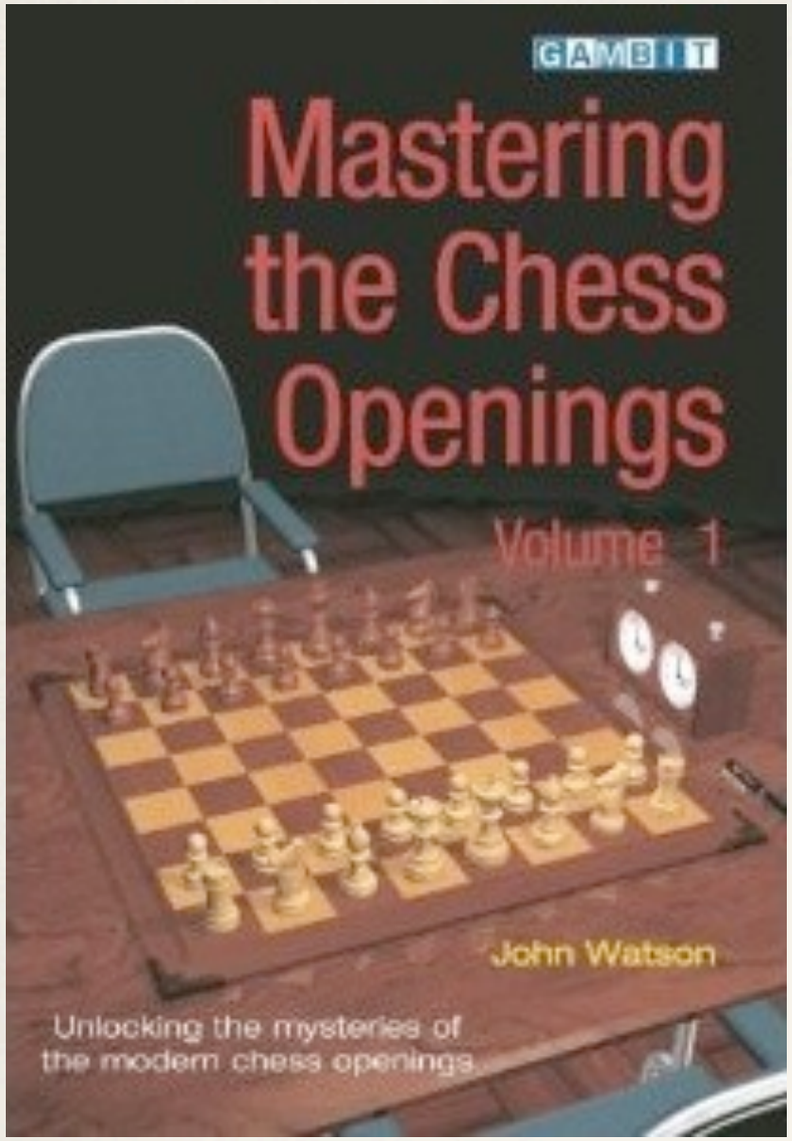
The Aspects of Programming

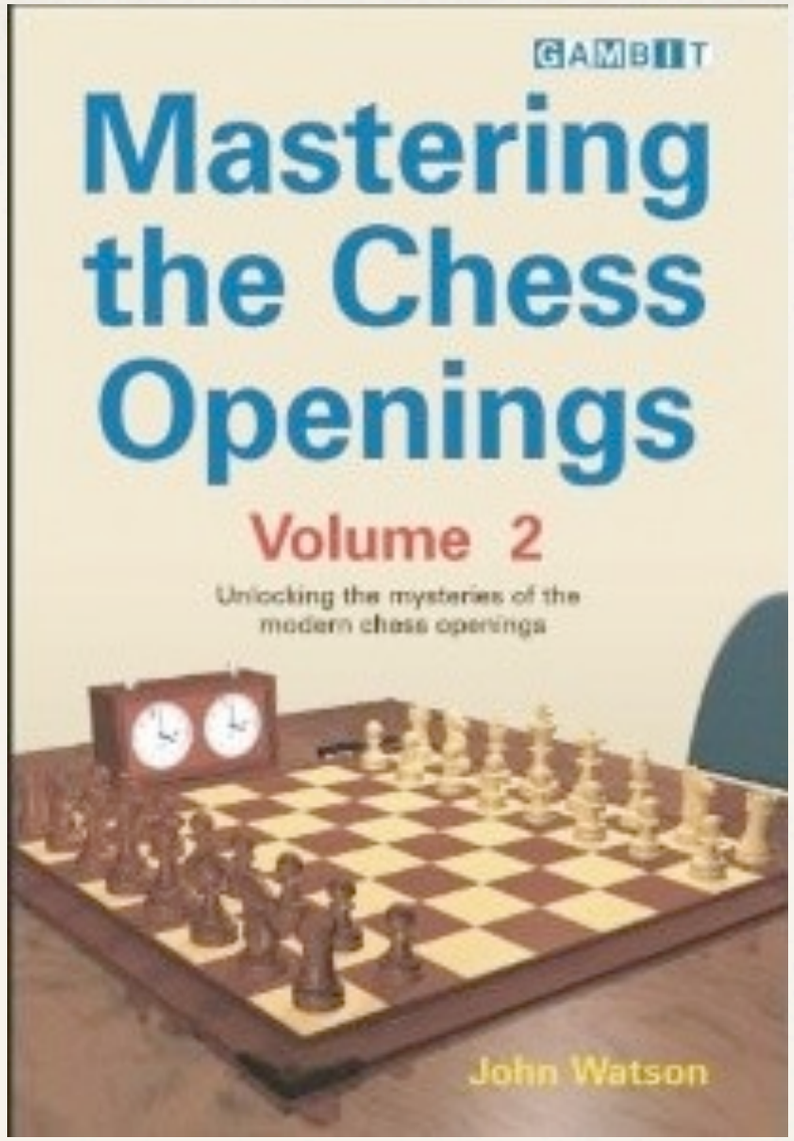
Or “What We Can Learn From the Chess Masters”

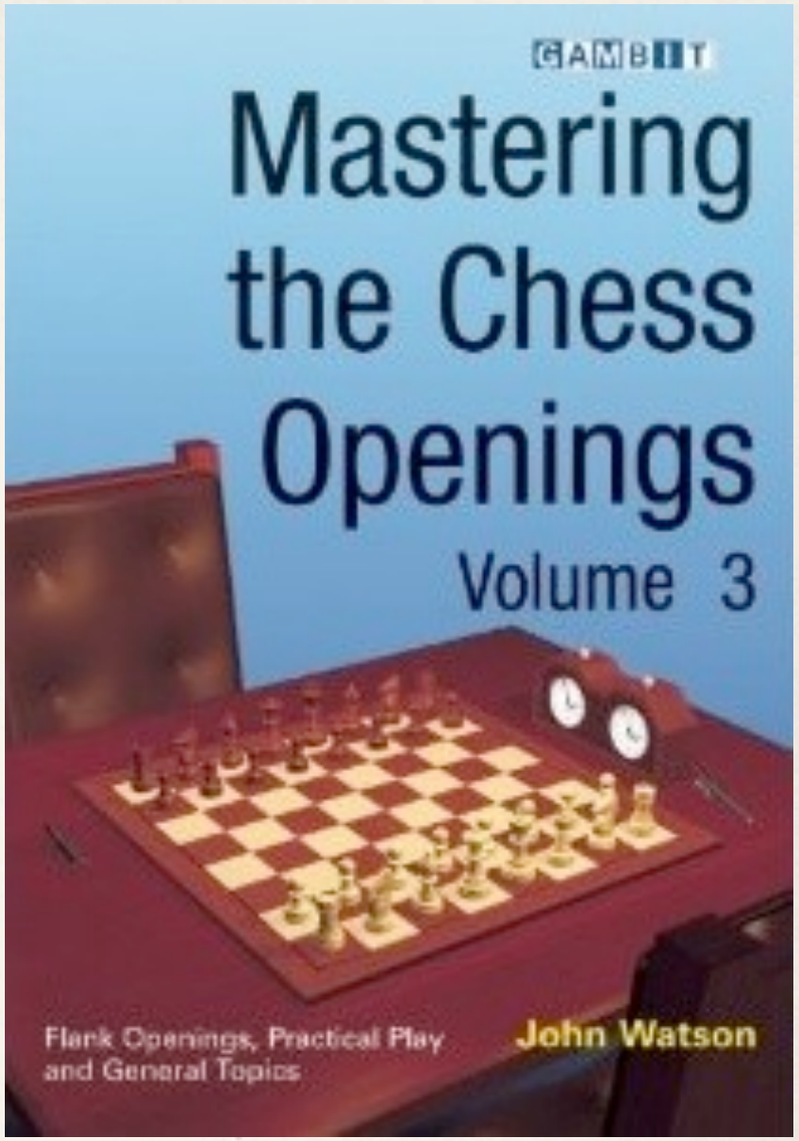
Warning: Regular Expressions Inside!

James Edward Gray II

- I was Highgroove employee #2 or #3
- I'm a regular on The Ruby Rogues podcast
- I've written a lot of code and documentation for Ruby, including CSV
- I also play some chess







The Chess Openings

All three volumes of them

You Can't Memorize This!

Sound Like Anything Else You Know?

What the Chess Masters Do

- Be familiar with as many openings as possible
- Know a few openings as well as anyone in the world

Can We Apply That to Programming?

A Suggested Strategy

- Be familiar with as many aspects of programming as possible
- Know some aspects as well as any programmer

Think of the Value

- To you
  - Having strong areas to lean on often helps with the scary tasks
  - It's possible to substitute knowledge groups in some areas
- To your team
  - How would you like to have Avdi Grimm around when it's time for some serious error handling?
  - Go for areas where the team is currently light

I Do This (By Accident)

My Familiarity With Ruby

- Running the Ruby Quiz for three years meant that I deciphered multiple clever Ruby programs each week
- Highgroove was fantastic for learning how to build applications: I worked through new challenges pretty much daily
- Being on the Ruby Rogues means I have to learn enough about a new Ruby topic each week to be able to credibly discuss it

Some Things I Really Know

- Non-blocking, multiplexing servers
  - I was obsessed with MUD's
  - Lightly useful
- Multilingualization (M17n)
  - I reverse engineered the initial m17n to document it
  - I wrote the first serious m17n-savvy library: CSV
  - Very useful

I Also Know Regular Expression

- My early programming jobs involved cleaning up some scary data from a black box system
- I was a big Perl junkie
- I’ve done a lot of work with TextMate’s (regex-based) parser
- Other programmers seem afraid of them, which pushed me harder
- This has proven crazy useful to me

Let’s See How Deep Your Regex Knowledge Goes…

```ruby
str = "Some long words and some shorter words."
str.scan(/\w*or\w*/) do |word|
  puts "#{word} has an or in it."
end
# >> words has an or in it.
# >> shorter has an or in it.
# >> words has an or in it.

list = str.split
p list.grep(/\A.{4}\z/)  # >> ["Some", "long", "some"]

i = str.rindex(/w\S*/)
p i                      # >> 33
puts str[i..-1]          # >> words.
```

Do You Know Which Ruby Methods Can Take a Regex?

Ruby has a lot of regex integration (and remember methods like gsub() take a block)

```ruby
str = "Some long words and some shorter words."
str.scan(/\w*or\w*/) do |word|
  puts "#{word} has an or in it."
end
# >> words has an or in it.
# >> shorter has an or in it.
# >> words has an or in it.

list = str.split
p list.grep(/\A.{4}\z/)  # >> ["Some", "long", "some"]

i = str.rindex(/w\S*/)
p i                      # >> 33
puts str[i..-1]          # >> words.
```

Do You Know Which Ruby Methods Can Take a Regex?

Ruby has a lot of regex integration (and remember methods like gsub() take a block)

```ruby
str = "Some long words and some shorter words."
str.scan(/\w*or\w*/) do |word|
  puts "#{word} has an or in it."
end
# >> words has an or in it.
# >> shorter has an or in it.
# >> words has an or in it.

list = str.split
p list.grep(/\A.{4}\z/)  # >> ["Some", "long", "some"]

i = str.rindex(/w\S*/)
p i                      # >> 33
puts str[i..-1]          # >> words.
```

Do You Know Which Ruby Methods Can Take a Regex?

Ruby has a lot of regex integration (and remember methods like gsub() take a block)

```ruby
str = "Some long words and some shorter words."
str.scan(/\w*or\w*/) do |word|
  puts "#{word} has an or in it."
end
# >> words has an or in it.
# >> shorter has an or in it.
# >> words has an or in it.

list = str.split
p list.grep(/\A.{4}\z/)  # >> ["Some", "long", "some"]

i = str.rindex(/w\S*/)
p i                      # >> 33
puts str[i..-1]          # >> words.
```

Do You Know Which Ruby Methods Can Take a Regex?

Ruby has a lot of regex integration (and remember methods like gsub() take a block)

```ruby
nums = "one two three four five"
puts nums[/t\w*/]  # >> two

puts nums[/t(\w*)/, 1]  # >> wo
# similar to:  nums =~ /t(\w*)/ && $1

p nums[/z(\w*)/, 1]  # >> nil
```

ALL The Methods?

This is often more helpful than =~

```ruby
nums = "one two three four five"
puts nums[/t\w*/]   # >> two

puts nums[/t(\w*)/, 1]   # >> wo
# similar to:  nums =~ /t(\w*)/ && $1

p nums[/z(\w*)/, 1]   # >> nil
```

ALL The Methods?

This is often more helpful than =~

Crazy Useful

```ruby
nums = "one two three four five"
puts nums[/t\w*/]   # >> two

puts nums[/t(\w*)/, 1]   # >> wo
# similar to:  nums =~ /t(\w*)/ && $1

p nums[/z(\w*)/, 1]   # >> nil
```

ALL The Methods?

This is often more helpful than =~

```ruby
nums = "one two three four five"
puts nums[/t\w*/]   # >> two

puts nums[/t(\w*)/, 1]   # >> wo
# similar to:  nums =~ /t(\w*)/ && $1

p nums[/z(\w*)/, 1]   # >> nil
```

ALL The Methods?

This is often more helpful than =~

```
order = <<END_ORDER
Item            |   Price | Quantity |   Total
---------------+---------+----------+-------
Pricey Thingy | $10.00 |        2 | $20.00
Cheap Thingy   |   $5.00 |        1 |   $5.00
---------------+---------+----------+-------
Total          |         |          | $25.00
END_ORDER

puts order[/\S+$/]    # >> Total
puts order[/\S+\Z/]   # >> $25.00
```

Do You Know the Anchors?

This helps optimize regexen and match the desired content

Maximum Useful

```
order = <<END_ORDER
Item            |   Price | Quantity |   Total
----------------+---------+----------+--------
Pricey Thingy   | $10.00  |        2 | $20.00
Cheap Thingy    |  $5.00  |        1 |  $5.00
----------------+---------+----------+--------
Total           |         |          | $25.00
END_ORDER

puts order[/\S+$/]     # >> Total
puts order[/\S+\Z/]    # >> $25.00
```

Do You Know the Anchors?

This helps optimize regexen and match the desired content

```
order = <<END_ORDER
Item           |  Price | Quantity |  Total
---------------+--------+----------+-------
Pricey Thingy | $10.00 |        2 | $20.00
Cheap Thingy   |  $5.00 |        1 |  $5.00
---------------+--------+----------+-------
Total          |        |          | $25.00
END_ORDER


puts order[/\S+$/]    # >> Total
puts order[/\S+\Z/]   # >> $25.00
```

Do You Know the Anchors?

This helps optimize regexen and match the desired content

```
bad_data = <<END_BAD
body { padding: 10px; }
p { color: #FF0000; }
<p>Some HTML</p>
END_BAD

bad_data.gsub!(/\G\w+\s*\{[^}]*\}\s*/, "")

puts bad_data  # >> <p>Some HTML</p>
```

ALL The Anchors?

I never see this one in the wild (and lookup \b if you don't know it)

```
bad_data = <<END_BAD
body { padding: 10px; }
p { color: #FF0000; }
<p>Some HTML</p>
END_BAD

bad_data.gsub!(/\G\w+\s*\{[^}]*\}\s*/, "")

puts bad_data   # >> <p>Some HTML</p>
```

ALL The Anchors?

I never see this one in the wild (and lookup \b if you don't know it)

```ruby
puts (1..100).map(&:to_s)
              .grep(/\A0*(?:2[0-5]|1\d|[1-9])\z/)
              .last
# >> 25

csv = <<END_CSV
a field,"a ""quoted"" field"
END_CSV
puts csv[/"(?:""|[^"]*)+"/]
# >> "a ""quoted"" field"
```

Do You Know The Common Patterns?

It's very useful to be able to match against a set of options

Very Useful

```ruby
puts (1..100).map(&:to_s)
              .grep(/\A0*(?:2[0-5]|1\d|[1-9])\z/)
              .last
# >> 25

csv = <<END_CSV
a field,"a ""quoted"" field"
END_CSV
puts csv[/"(?:""|[^"]*)+"/]
# >> "a ""quoted"" field"
```

Do You Know The Common Patterns?

It's very useful to be able to match against a set of options

Very Useful

```ruby
puts (1..100).map(&:to_s)
              .grep(/\A0*(?:2[0-5]|1\d|[1-9])\z/)
              .last
# >> 25

csv = <<END_CSV
a field,"a ""quoted"" field"
END_CSV
puts csv[/"(?:""|[^"]*)+"/]
# >> "a ""quoted"" field"
```

Do You Know The Common Patterns?

It's very useful to be able to match against a set of options

Very Useful

```ruby
num = "$10000.00"
puts num.reverse
        .gsub(/(?!\d*\.)\d{3}/, '\0,')
        .reverse
# >> $10,000.00
```

Do You Know How to Use a Look-around Assertion?

This is great for refining matches

Pretty Useful

```ruby
num = "$10000.00"
puts num.reverse
        .gsub(/(?!\d*\.)\d{3}/, '\0,')
        .reverse
# >> $10,000.00
```

Do You Know How to Use a Look-around Assertion?

This is great for refining matches

Pretty Useful

```ruby
NAME_RE = /(?<last>\w+),\s*(?<first>\w+)/
DATA    = "Gray, James"

if DATA =~ NAME_RE
  puts $~[:first]   # >> James
end

# my favorite again
puts DATA[NAME_RE, :last]   # >> Gray

# party trick
if /(?<last>\w+),\s*(?<first>\w+)/ =~ DATA
  p [first, last]   # >> ["James", "Gray"]
end
```

Do You Know You Can Use Names Instead of Numbers?

This can really boost code clarity

```ruby
NAME_RE = /(?<last>\w+),\s*(?<first>\w+)/
DATA    = "Gray, James"

if DATA =~ NAME_RE
  puts $~[:first]   # >> James
end

# my favorite again
puts DATA[NAME_RE, :last]   # >> Gray

# party trick
if /(?<last>\w+),\s*(?<first>\w+)/ =~ DATA
  p [first, last]   # >> ["James", "Gray"]
end
```

Do You Know You Can Use Names Instead of Numbers?

This can really boost code clarity

```ruby
NAME_RE = /(?<last>\w+),\s*(?<first>\w+)/
DATA    = "Gray, James"

if DATA =~ NAME_RE
  puts $~[:first]   # >> James
end

# my favorite again
puts DATA[NAME_RE, :last]   # >> Gray

# party trick
if /(?<last>\w+),\s*(?<first>\w+)/ =~ DATA
  p [first, last]   # >> ["James", "Gray"]
end
```

Do You Know You Can Use Names Instead of Numbers?

This can really boost code clarity

```ruby
NAME_RE = /(?<last>\w+),\s*(?<first>\w+)/
DATA    = "Gray, James"

if DATA =~ NAME_RE
  puts $~[:first]   # >> James
end

# my favorite again
puts DATA[NAME_RE, :last]   # >> Gray

# party trick
if /(?<last>\w+),\s*(?<first>\w+)/ =~ DATA
  p [first, last]   # >> ["James", "Gray"]
end
```

Do You Know You Can Use Names Instead of Numbers?

This can really boost code clarity

```ruby
NAME_RE = /(?<last>\w+),\s*(?<first>\w+)/
DATA    = "Gray, James"

if DATA =~ NAME_RE
  puts $~[:first]   # >> James
end

# my favorite again
puts DATA[NAME_RE, :last]   # >> Gray

# party trick
if /(?<last>\w+),\s*(?<first>\w+)/ =~ DATA
  p [first, last]   # >> ["James", "Gray"]
end
```

Do You Know You Can Use Names Instead of Numbers?

This can really boost code clarity

```ruby
NUM_REGEX = / \A          # the start of the number
              0*          # zero or more leading zeros
              (?:
                2[0-5]    # 20-25
                |         # ...or...
                1\d       # 10-19
                |         # ...or...
                [1-9]     # 1-9
              )
              \z          # the end of the number
              /x
puts (1..100).map(&:to_s).grep(NUM_REGEX).last   # >> 25
```

Do You Know That a Regex Can Be Readable?

Another great resource for code clarity

Not Always Needed, But Invaluable For Complex Expressions

```ruby
NUM_REGEX = / \A          # the start of the number
              0*          # zero or more leading zeros
              (?:
                2[0-5]    # 20-25
                |         # ...or...
                1\d       # 10-19
                |         # ...or...
                [1-9]     # 1-9
              )
              \z          # the end of the number
              /x

puts (1..100).map(&:to_s).grep(NUM_REGEX).last   # >> 25
```

Do You Know That a Regex Can Be Readable?

Another great resource for code clarity

Not Always Needed, But Invaluable For Complex Expressions

```
html = <<END_HTML
<!-- balanced tags: we'll match the outer div with its content -->
<div class="article">
  <p>paragraph one</p>
  <p>paragraph two</p>
</div>
END_HTML
puts html[ %r{
  (?<tag>                    # a named group
    <(?<name>\w+)[^>]*>      # an opening tag
      (?:
        \g<tag>              # recursion:  another full tag
        |                    # ...or...
        [^<]*+               # some content (non-backtracking for speed)
      )+
    </\k<name+0>>            # the matching end tag
  )
}x ]
```

Do You Know How to Write a Recursive Regex?

This makes some extremely complex matches possible

Rarely Needed, But Powerful

```
html = <<END_HTML
<!-- balanced tags: we'll match the outer div with its content -->
<div class="article">
  <p>paragraph one</p>
  <p>paragraph two</p>
</div>
END_HTML
puts html[ %r{
  (?<tag>                    # a named group
    <(?<name>\w+)[^>]*>      # an opening tag
      (?:
        \g<tag>              # recursion:  another full tag
        |                    # ...or...
        [^<]*+               # some content (non-backtracking for speed)
      )+
    </\k<name+0>>            # the matching end tag
  )
}x ]
```

Do You Know How to Write a Recursive Regex?

This makes some extremely complex matches possible

Rarely Needed, But Powerful

```
html = <<END_HTML
<!-- balanced tags: we'll match the outer div with its content -->
<div class="article">
  <p>paragraph one</p>
  <p>paragraph two</p>
</div>
END_HTML
puts html[ %r{
  (?<tag>                    # a named group
    <(?<name>\w+)[^>]*>      # an opening tag
      (?:
        \g<tag>              # recursion:  another full tag
        |                    # ...or...
        [^<]*+               # some content (non-backtracking for speed)
      )+
    </\k<name+0>>            # the matching end tag
  )
}x ]
```

Do You Know How to Write a Recursive Regex?

This makes some extremely complex matches possible

Rarely Needed, But Powerful

```ruby
html = <<END_HTML
<!-- balanced tags: we'll match the outer div with its content -->
<div class="article">
  <p>paragraph one</p>
  <p>paragraph two</p>
</div>
END_HTML
puts html[ %r{
  (?<tag>                    # a named group
    <(?<name>\w+)[^>]*>      # an opening tag
      (?:
        \g<tag>              # recursion:  another full tag
        |                    # ...or...
        [^<]*+               # some content (non-backtracking for speed)
      )+
    </\k<name+0>>            # the matching end tag
  )
}x ]
```

Do You Know How to Write a Recursive Regex?

This makes some extremely complex matches possible

Rarely Needed, But Powerful

```ruby
html = <<END_HTML
<!-- balanced tags: we'll match the outer div with its content -->
<div class="article">
  <p>paragraph one</p>
  <p>paragraph two</p>
</div>
END_HTML
puts html[ %r{
  (?<tag>                    # a named group
    <(?<name>\w+)[^>]*>      # an opening tag
      (?:
        \g<tag>              # recursion:  another full tag
        |                    # ...or...
        [^<]*+               # some content (non-backtracking for speed)
      )+
    </\k<name+0>>            # the matching end tag
  )
}x ]
```

Do You Know How to Write a Recursive Regex?

This makes some extremely complex matches possible

Rarely Needed, But Powerful

What Do You Know as Good as Any Programmer?

Things I Don’t See Enough

- Algorithm and data structure junkies
- ncurses gurus
- C extension authors
- Mutiprocessing/multithreading pros
- Raspberry Pi hobbyists
- Mathletes
- …

Thanks

Questions?