Ruby Control Flow

CSCI400

29 August 2017

Color Key

- Clickable URL link
- Write down an answer to this for class participation
- Just a comment – don't confuse with yellow

Standard Control Flow

- Selection statements
- Iterative statements
- Unconditional branching
- Not covered: Guarded commands

Conditionals

```
if expr1 # newline after expr
    # code
elsif expr2
    # code
...
else
    # code
end # `end` always required
```

Conditional is executed if `expr` is *not* `false` or `nil`

Conditional return value

Return value is last expression executed *or* `nil`

```
x = 5
# note the lack of `return`
name = if x == 1 then "Cyndi" else "Nancy" end
puts name
```

Expression Modifier

`if expr then code end` equivalent to `code if expr`

Best practice: use latter form when `expr` is trivial or normally true

Perl also has this syntax

Other conditionals: `unless`

```
unless expr
    # code
end
# or
code unless expr
```

Other conditionals: `case/when`

```
tax = case income
    when 0..7550
        income * 0.1
    when 7550..30650
        income * 0.15
    when 3065..50000
        income * 0.25
    else
        income * 0.4
end
```

Compare to ‘switch‘; consider readability

Iteration

`while`	`until`

`while expr do`
`    # code`
`end`
`# or`
`code while expr`
	`until expr do`
	`    # code`
	`end`
	`# or`
	`code until expr`

Pascal had `repeat...until...`

More Iteration

```
# do is optional, can use newline
for var in collection do
    # code
end
```

```
hash.each do |key, value|
    puts "#{key} => #{value}"
end
```

Iterators

- `<int>.times`
  - `2.times { puts "again!" }`
- `<enumerable>.each`
  - `array.each { |x| puts x }`
- `<enumerable>.map`
  - `[5, 10, 15].map { |x| x * x * x }`
- `<int>.upto`, `<int>.downto`
  - `factorial = 1; 2.upto(20) { |x| factorial *= x }`
- Make use of `yield` (next slide)

`yield`

`yield` temporarily returns control from iterator to calling method

Exercise

- Trace the code on the next two slides
- Format is flexible
  - Draw arrows, etc. Just show you understand
- **Discuss** when/why might this be useful?
  - We'll discuss as a class

yield example (1)

`yield` temporarily returns control from iterator to calling method

```
def test
    puts "You are in the method"
    yield
    puts "You are back in the method"
    yield
end

test { puts "You are in the block" }
```

Method must be invoked with a block (which is the code that is `yield`ed to)

`yield` example (1)

Result of running code on previous slide:

```
You are in the method
You are in the block
You are back in the method
You are in the block
```

yield example (2)

```ruby
def test
    yield 5
    puts "You are in the method 'test'"
    yield 100
end

test { |i| puts "You are in the block: #{i}" }
```

yield example (2)

```
You are in the block: 5
You are in the method test
You are in the block: 100
```

Java: caller controls iteration

Ruby: iterator controls iteration

Discussion

When/why might `yield` be useful?

`yield` in-class challenge

- Write code similar to '`yield` example 2' that:
  - Displays the modulo 15 of all integers within `[100, 91]`
  - Your `yield` expression should provide *two* values
- **Hint:** in the output below, what changes and what stays the same?

```
100 modulo 15 is 10
99 modulo 15 is 9
98 modulo 15 is 8
97 modulo 15 is 7
96 modulo 15 is 6
```

- Nothing to submit

Language Design: Importance of Blocks

Read this