Lesson 3

Making decisions & reusing code

- What if anything must be done before printing a number?
- How do you display variable values within a string?
- How do you access values within an array?
- How are arrays and hashes different?
- What is a range?

Objectives

- Conditional love
- Branch execution
- Custom methods

Logic

==	Same value?	puts a == b	
!=	Different value?	puts a != b	
>	Greater than?	puts a > b	
<	Less than?	puts a < b	
>=	Greater than or same?	puts a >= b	
<=	Less than or same?	puts a <= b	
<=>	Comparision	puts a <=> b	1 greater, 0 equal, -1 less than
.eql?	Same value and type?	puts a.eql?(b)	1.eql?(1.0) => false

if statements test conditions

Multi-line logical expressions use if and end

```
a = 5
if a > 4
    puts "why yes it is"
end
=> why yes it is
```

Single-line logical expressions use if, then, and end

```
a = 5
if a > 4 then puts "indeed it's true" end
=> indeed it's true

puts “indeed it’s true” if a > 10
```

You can test if both or all of a set of conditions are true

&&: is true if both left and right conditions are true

```
a, b = 4, "Ginger"
if a == 4 && b == "Ginger"
    puts "yes, both are true"
end
=> yes, both are true
```

||: is true if either left or right conditions are true

```
a, b = 4, "Ginger"
if a == 4 || b == 2
    puts "well, one of them is true"
end
=> well, one of them is true
```

You can do multiple conditions too!

Brackets help make your logic clear

```
a, b, c = 4, 7, 8
if (a == 4 && b == 3) || c == 8
    puts "well, c is 8 so the whole thing's true"
end
=> well, c is 8 so the whole thing's true
```

Test for falsity with !, not, or unless

```
role = "user"

if !(role == "admin")
    puts "it's true, role is not admin"
end
=> it's true, role is not admin

if not role == "admin"
    puts "no, role is still not admin"
end
=> no, role is still not admin

unless role == "admin"
    puts "role is still not admin"
end
=> role is still not admin
```

Loops can be controlled during iteration

break: stop executing the whole loop and continue just after it
next: stop the current iteration and go on to the next one
redo: repeat the current iteration
retry: start the whole loop over from the beginning

```
flintstones = ["Fred", "Wilma", "Barney", "Betty"]

flintstones.each do |name|
    next unless name == "Barney" || name == "Wilma"
    puts name
end
=> Wilma
=> Barney
```

Exercise: comparing values in conditional statements

If a condition is false, you can ask alternatives for with elsif and set a default with else

```
user_score, dealer_score = 7, 11

if user_score > dealer_score
    puts "user wins"
elsif user_score == dealer_score
    puts "it's a tie"
else
    puts "user loses"
end
=> user loses
```

Both if and elsif support multiple conditions

```
if user_score > dealer_score
    puts "user wins"
elsif user_score == dealer_score && choice == "S"
    puts "it's a tie"
else
    puts "user loses"
end
```

The case, when and else statements testing multiple values and provide a default

```
choice = $stdin.gets.chomp

result = case choice
    when "H" then "Hit"
    when "S" then "Stand"
    when "Q" then "Quit"
    else "Invalid Choice"
end

puts result
```

Exercise:

Branching execution based on conditions

A method is a named code block wrapped by def and end which you may execute by name

In other languages, methods may be known as functions, operations, procedures, sub-routines, etc..

Methods may or may not return a value

In Ruby, the last expression value is returned whether or not you use the return statement

```
def say_hello
   return "Hello!" # you can omit return
end

puts say_hello
=> Hello!
```

Parameters may be defined so that arguments may be passed to your method

Brackets are optional around your arguments

```
def add_numbers(a, b)
    a + b
end

puts add_numbers(2, 3)
=> 5

puts add_numbers 5, 7
=> 12
```

A value assigned in the parameter list is treated as its default

```
def add_numbers(a, b=5)
    a + b
end

puts add_numbers(2)
=> 7

puts add_numbers(2, 9)
=> 11
```

Parameters preceded by * pass any number of values to the method as an array

```
def add_several_numbers(*a)
    result = 0
    a.each do |num|
        result = result + num
    end
    return result
end

puts add_several_numbers(3, 6, 9)
=> 18
```

The implications of combining parameter types provides an interesting discussion here:
http://www.skorks.com/2009/08/method-arguments-in-ruby/

How does a method access variables outside the method?

Usually it doesn't, a method defines its own variable scope

```
colors = ["red", "green", "blue]
def get_color(index)
    return colors(index)
end

puts get_color(1)
=> Errors because colors array not in scope
within get_color method
```

Values to be used in a method are normally passed in as parameters

```
colors = ["red", "green", "blue"]

def get_color(colors, index)
    return colors[index]
end
```

```
puts get_color(colors, 1)
=> green
```

get_color method now works because colors array passed to method along with index

global, class, and instance variable scopes which could change this behavior are discussed later

Exercise:

Writing and calling custom methods

Lab:
Making decisions
& reusing code