AIMS FOR TODAY

- Hard core review of last week's Ruby booleans
- Ruby's loops and collections

LOGICAL OPERATORS

Operator	Description	Example (a =4 and b= 2)
==	Equal	a == b *false*
!=	Not Equal	a != b *true*
>	Greater than	a > b *true*
<	Less than	a < b *true*
>=	Greater than or equal to	a <= b *false*
<=	Less than or equal to	a <= b *false*
⇔	same value? return 0 less than? return -1 greater than? return 1	a <=> b *1*
.eql?	same value and same type?	1.eql?(1.0) *false*

- These return true or false

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(A and B) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A or B) is true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

&&

- Both sides must be true for an and statement to be true

- If both are true and non zero, this evaluates to true

- `3 == 1 + 2  &&  7 == 7 #=> true`

- `3 == 1 + 2  &&  7 == 8 #=> false`

||

- || means or. If one side or the other is true, then its true. If both are false, then false.

- If the left side statement is true, it doesn't bother to evaluate the right hand side.

- `3 == 1 + 2  ||  “foo” == “bar” #=> true`

- `3 == 1 + 2  ||  7 == 8 #=> true`

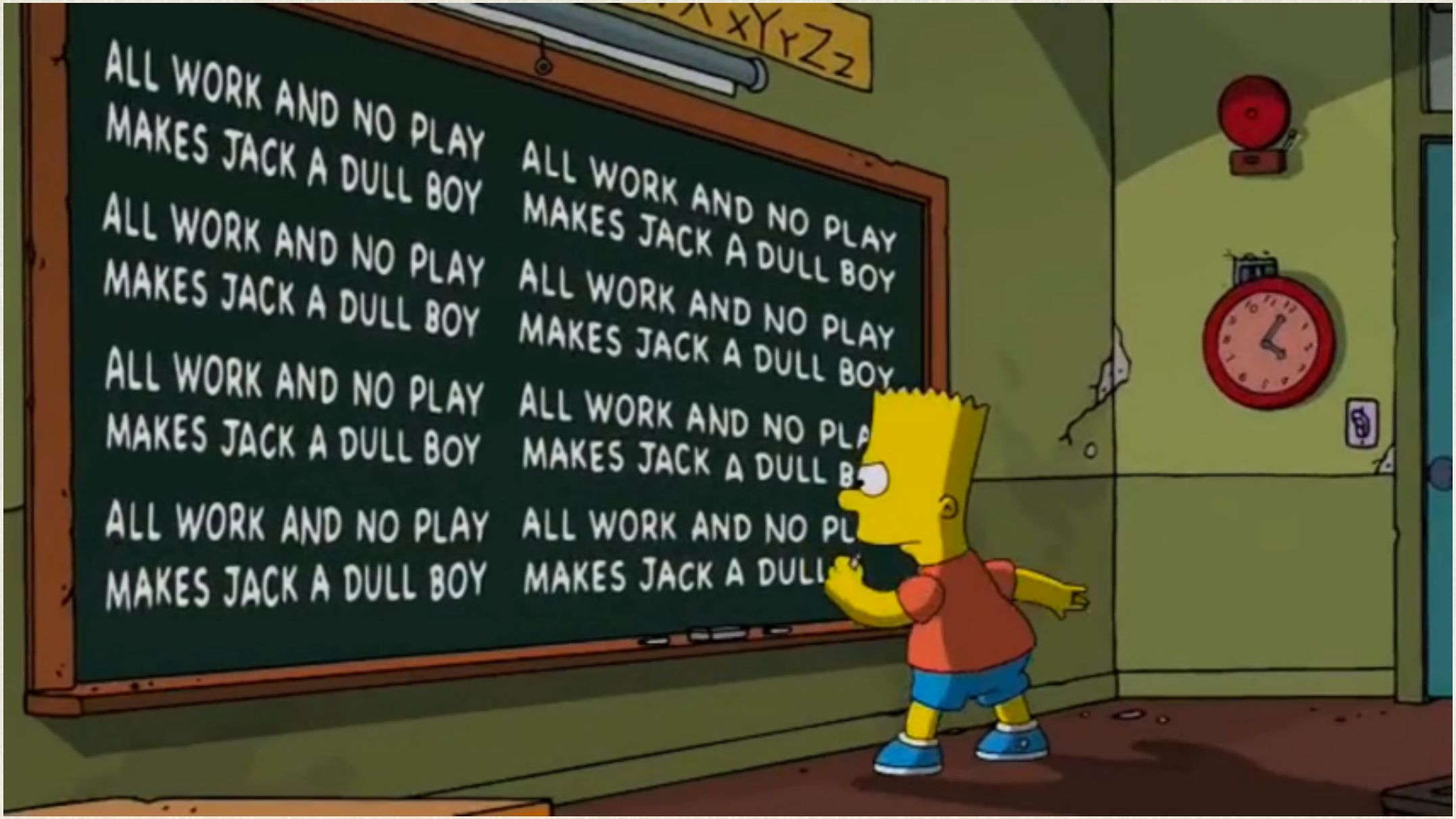
!

- Pronounced bang. Makes the evaluated statement the boolean opposite

- `!(“foo” == “bar”) #=> true`

- `!(“foo” == “foo”) #=> false`

TEDDIT CODE ALONG


ALL WORK AND NO PLAY
MAKES JACK A DULL BOY
ALL WORK AND NO PLAY
MAKES JACK A DULL BOY
ALL WORK AND NO PLAY
MAKES JACK A DULL BOY
ALL WORK AND NO PLAY
MAKES JACK A DULL BOY
ALL WORK AND NO PLAY
MAKES JACK A DULL BOY
ALL WORK AND NO PLAY
MAKES JACK A DULL BOY

REPETITION

LOOPS AND ITERATION

- Computers' tireless repetition and iteration of operations is what makes them so powerful.
- This way, we don't have to be like Bart Simpson, repeating ourselves. Let the computer do that.

An Example of a loop

```
# bart.rb

100.times do
  puts “I will not xerox my butt”
end
```

Will result in…

```
$ ruby bart.rb
I will not xerox my butt
I will not xerox my butt
I will not xerox my butt…
```

100 times!

A loop is a type of block

Start of block

```
100.times do
  puts “I will not xerox my butt”
end
```

End of block

The code that will run 100 times

BLOCKS

```
# loops.rb

3.times do
  puts “going...”
end
```

- The purple “do” and “end” indicate the start and end of something called a “block”. A block can be inside a method.
- A block is similar to a method, in that it contains a routine of code to be executed.
- The largest difference between blocks and methods are that they don’t have a “method definition”
  - i.e. You can’t execute a block via “do_something()”

UP TO X

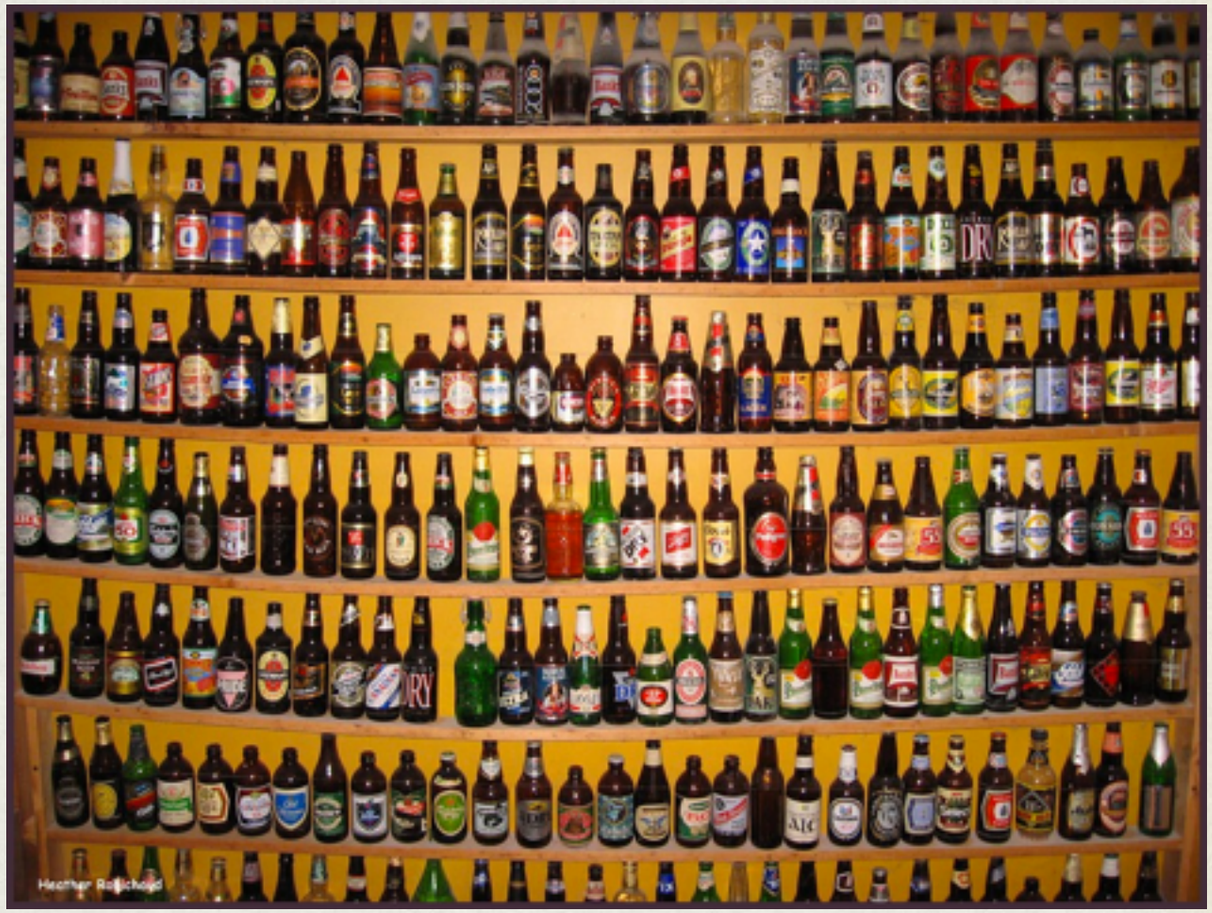
```
# upto_loop.rb

1.upto(10) do |num|
  puts “#{num}. going…”
end
```

- num starts at 1 and ends at 10
- It’s essentially a throw-away variable that changes at every cycle of the loop.
- num is only scoped within the block. It is meaningless outside of it.

```
irb(main):019:0> 1.upto(10) do |num|
irb(main):020:1* puts "#{num}. going..."
irb(main):021:1> end
1. going...
2. going...
3. going...
4. going...
5. going...
6. going...
7. going...
8. going...
9. going...
10. going...
```

THE MOST ANNOYING SONG EVER

- Partner up!
- Go to the docs/google.
- Look up downto
- Make Ruby sing the entire “99 bottles of beer on the wall song



CONDITIONAL LOOPS

These work like dynamic if statements

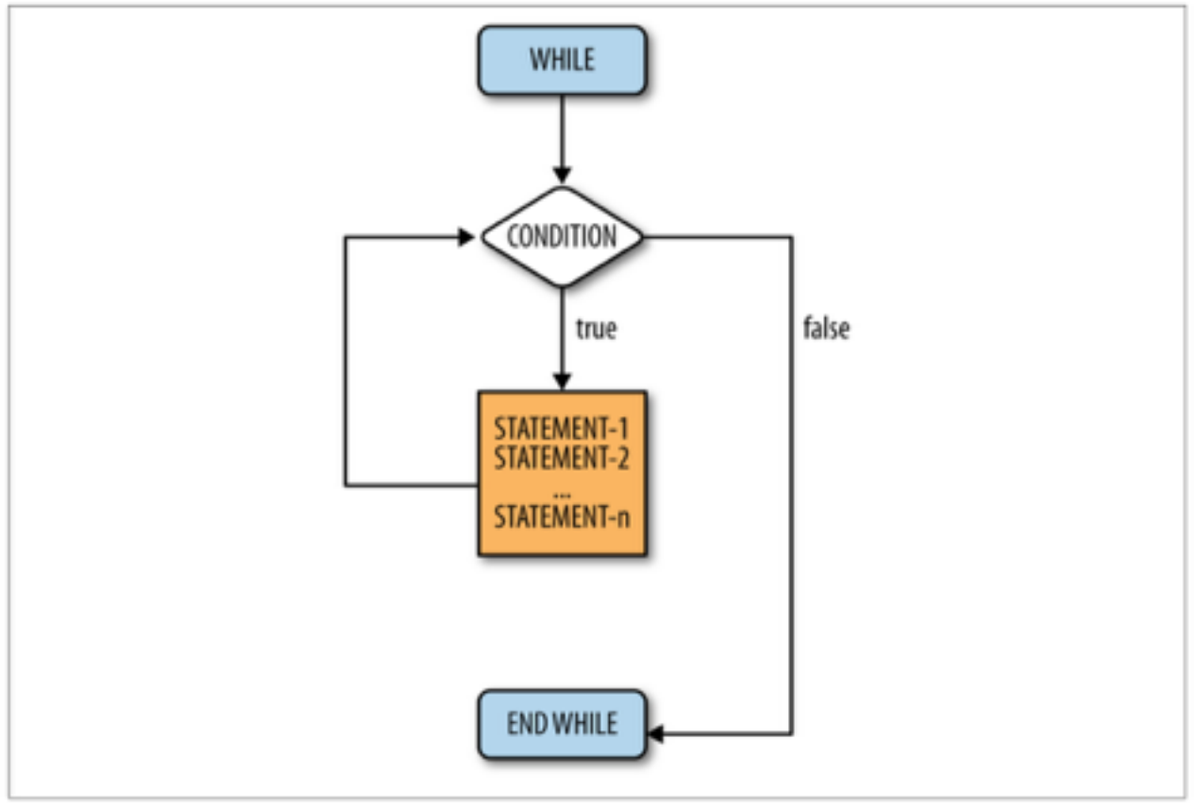
Conditional Loops

```
count = 10
while count > 0
    puts "Looping"
    count -=1
end

count = 10
until count < 1
    puts "Looping"
    count -= 1
end
```

CONDITIONAL LOOPS - WHILE

- As long as the condition is true, it will run forever

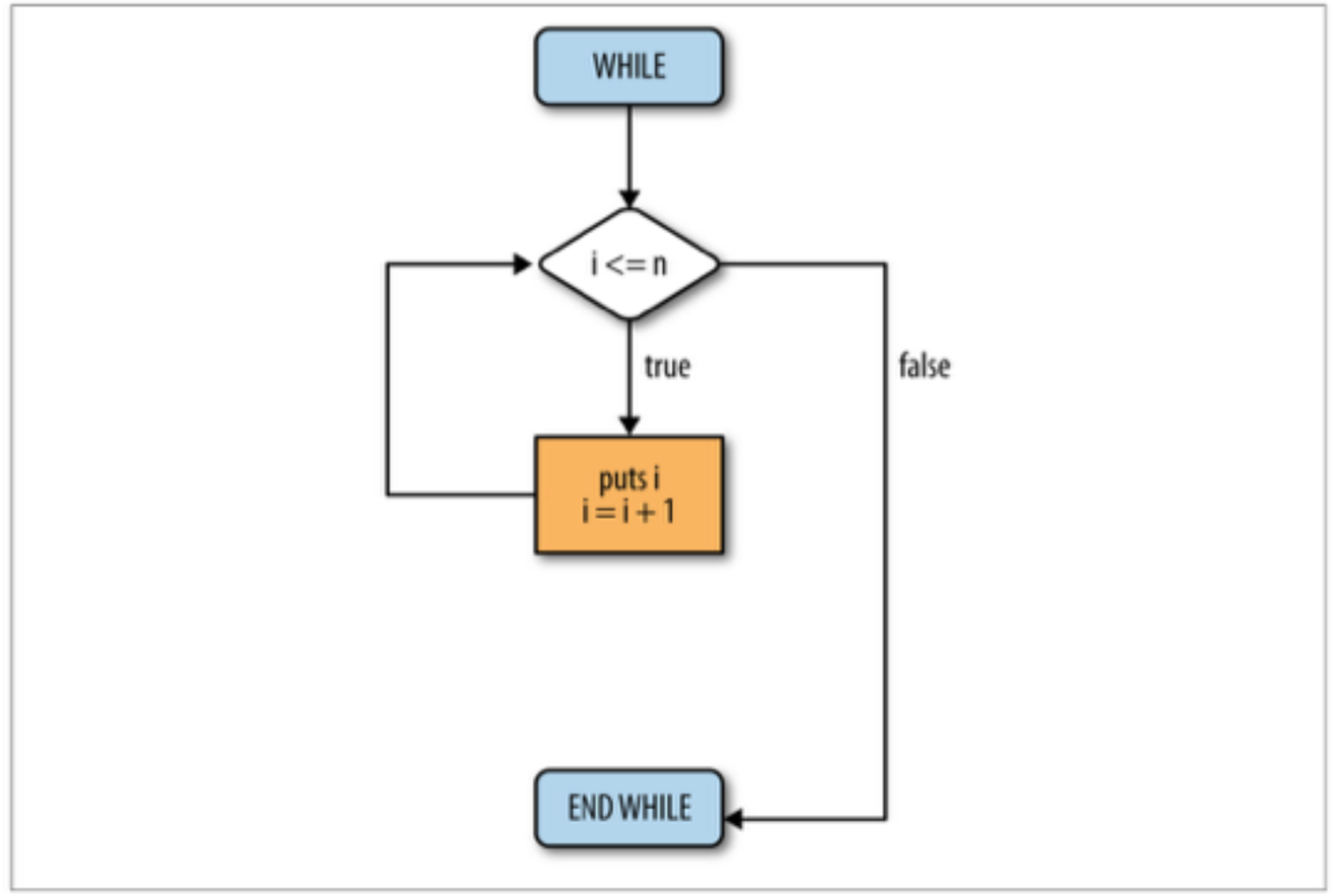
```ruby
while (condition)
  # statement 1
  # statement 2
  # ...
  # statement n
end
```



LOOPS MUST CONTAIN THE SEEDS OF THEIR OWN DESTRUCTION

- This while loop increments the incremented (i) so it ends

```
1 n = 5
2 i = 0
3 while (i <= n)
4   puts i
5   i = i + 1
6 end
```

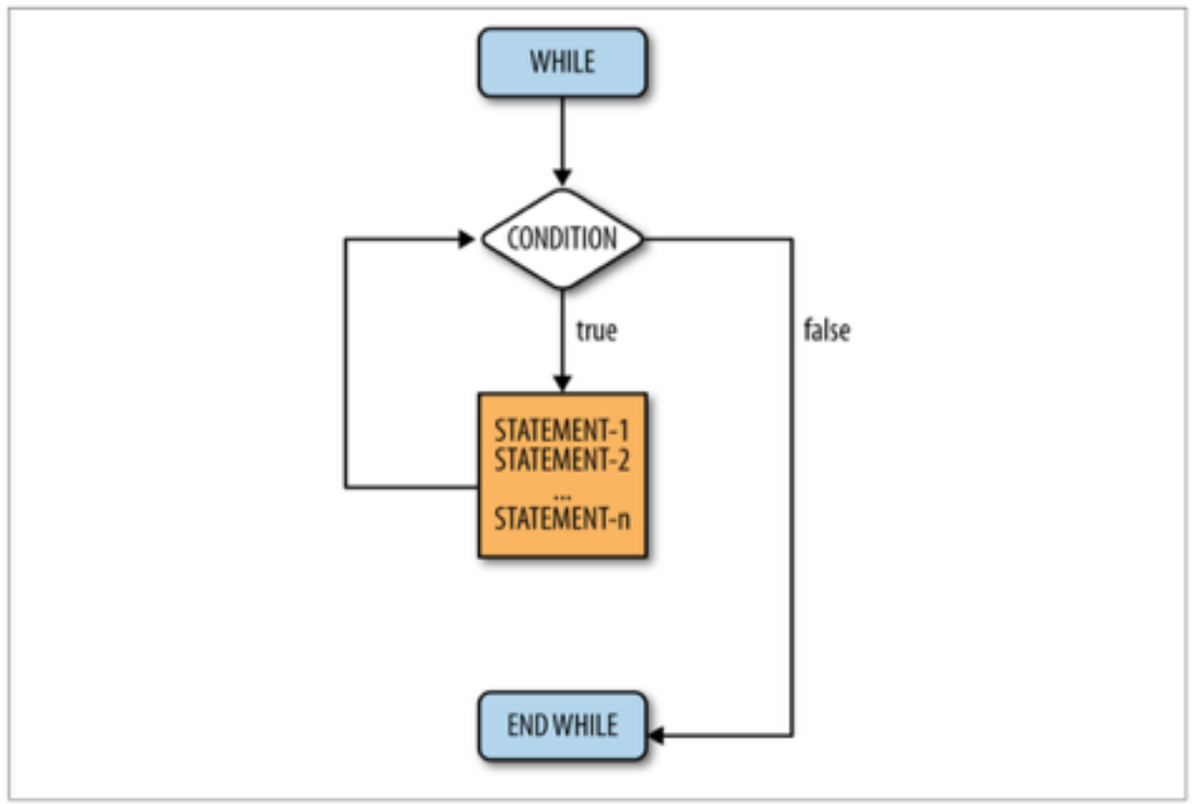


AVOIDING INFINITE LOOPS

- As long as the condition is true and never changes, it will run forever.

- If you don't, you have an infinite loop, which is very bad.

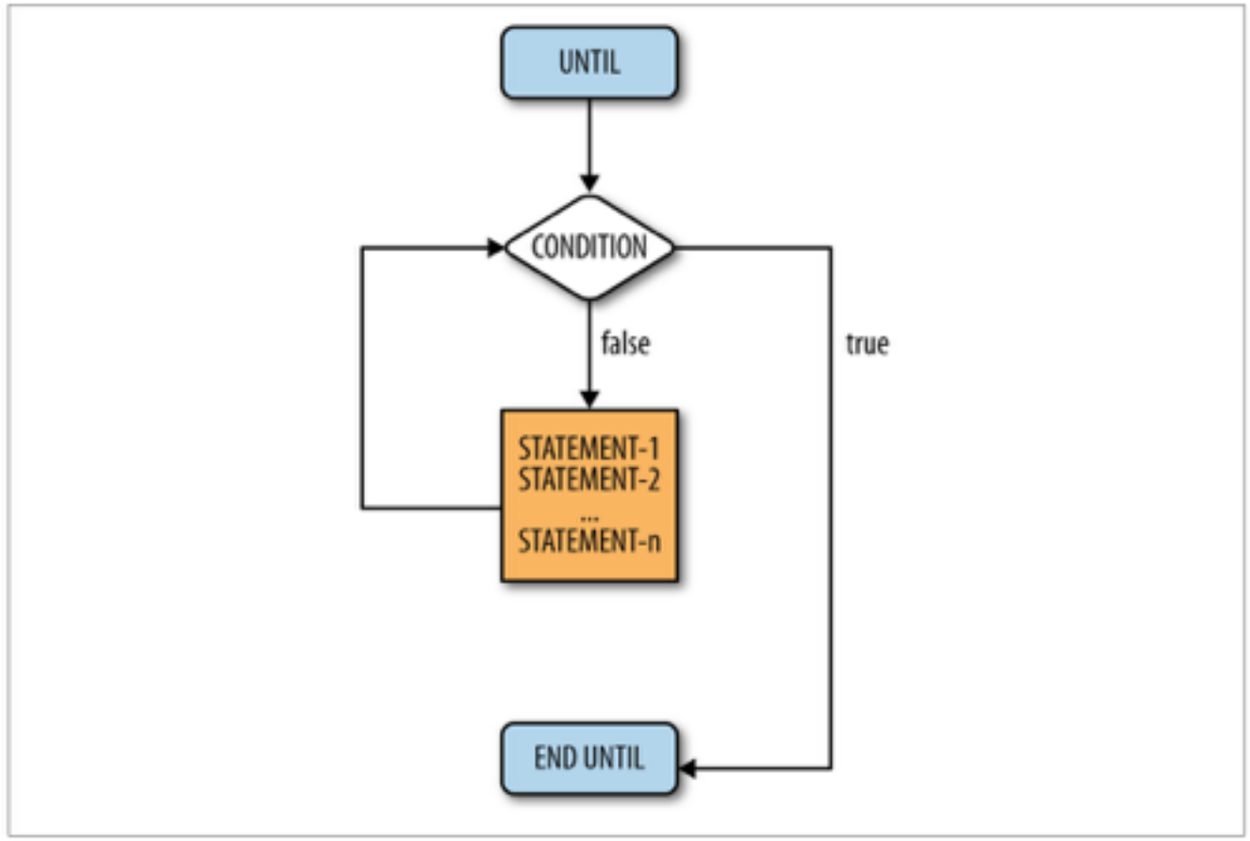
```ruby
while (condition)
  # statement 1
  # statement 2
  # ...
  # statement n
end
```



CONDITIONAL LOOPS - UNTIL

- Opposite of while.

```ruby
until (condition)
  # statement 1
  # statement 2
  # ...
  # statement n
end
```



REFACTORING

- Refactoring is taking working code, and tweaking it to make it run more efficiently or elegantly.

- Let's refactor the awesome song that you just made to use while and until loops.

- Try both!

BREAK FOR 5

ARRAYS

ARRAYS

- Arrays are an ordered collection of objects
- Arrays can contain any type of data (string, int, other arrays ...)
- Arrays can be extremely long (based on how you have ruby / your computer setup)
- Arrays can be iterated over to get every object individually

ARRAY SYNTAX

- Arrays start with [and end with], for example:
  - `[1, 2, 3]` generates an Array with 3 fixnums
  - `["a", "b", "c"]` generates an Array with 3 strings
  - `["1", 2, "abc"]` generates an Array with a mixed object collection
  - `["1", 2, , [1,2,3], "abc"]` generates an Array with a mixed object collection

ARRAY SYNTAX

- Accessing an Array uses [and] as well, but on the array itself.
- `people = [“Vincent”, “Eddie”, “Ali”]`
  - `me = people[0]`
  - `eddie = people[1]`
- All array’s start at 0, getting the first element must be 0
- You can use -1 to retrieve the last element
  - troll = people[-1]

CODE ALONG IN IRB

ARRAYS RECAP

- Arrays are ordered collections of data
- Each element can be accessed with an index
- The index is its *offset* from the first. The first one is 0, second is 1, and so on. You can use -1, -2 etc to count from the back.
- Arrays have lots of methods such as `.class`, `.size`, `.push`, `<< .unshift`, `.pop`, `.shift`, `.include?`, `.sort`, `.uniq`, `.shuffle`, `.join`, `.to_s`

1. Create fast_food.rb in your class3 folder
2. Create an array of fast food chains with 5 items
3. Display the the 2nd, and 5th restaurant in the array
4. Reverse the array, and display the 4th restaurant after it has been reversed
5. Remove the last element of the array and output that

HASHES

HASHES

- Often referred to as dictionaries
- Each entry in a hash needs a key and a value. The value can be any data type, including Hash.
- If you access a hash at a specific key, it will return the value at that key
- Think of a paper dictionary:
  - You know the word, but not what it means. It’s position in the
  - You look up the definition by going to where the word is in the book
  - You then read the definition

ACCESSING VALUES BY KEY

```
1  user = {"name" => "Vincent", "age" => 31, "home" =>
   "Brooklyn", "email" => "trivett@gmail.com"}
2
3  user["name"]
4  => "Vincent"
```

SETTING VALUES

```
1  user = {"name" => "Vincent", "age" => 31, "home" =>
   "Brooklyn", "email" => "trivett@gmail.com"}
2
3  user["job"] = "developer"
4
5  user
6  => {"name"=>"Vincent", "age"=>31, "home"=>"Brooklyn",
   "email"=>"trivett@gmail.com", "job"=>"developer"}
7
8
```

METHODS

```
1  user = {"name" => "Vincent", "age" => 31, "home" =>
   "Brooklyn", "email" => "trivett@gmail.com"}
2
3  user.has_key? "email"
4  =>true
5  user.keys
6  => ["name", "age", "home", "email"]
7
```

SYMBOLS

Symbols

New Ruby type

- A symbol is a special type of object in ruby, used extensively
- It is always preceded by a colon
- Cannot contain spaces or numbers
- Symbols are used because:
  - they are immutable and take less memory
  - they are easier to compare to other objects
  - they are cleaner in syntax
- Examples:
  - `:hello`
  - `:this_is_a_symbol`

SYMBOLS

Symbols

Primarily used as keys for hashes

```
ga_markets = {}
ga_markets = {:NYC => "New York City"}
ga_markets[:LA] = "Los Angeles"
ga_markets

>> {:NYC => "New York City", :LA => "Los Angeles"}
```

SYNTAX DIFFERENCES

Ruby 1.9+ Alternate Syntax

```
user = {:user_name => "SalmanAnsari", :email => "salman.ansari@gmail.com"}
# becomes

user = {user: "SalmanAnsari", email: "salman.ansari@gmail.com"}

# a little bit more concise
# more closely matches JSON format
# considered an 'alternate' syntax, not a replacement
```

Only works when the keys are symbols

Don’t get too hung up on syntax at the moment.

ACCESSING VALUES BY KEY (SYMBOL)

```
user = {name: "Vincent", age: 31, home: "Brooklyn", email: "trivett@gmail.com"}

user[:name]
=> "Vincent"

user[:age]
=> 31
```

LAB

- Teddit with hashes!

- Bonus: If you finish early, see if you can implement the cat/bacon based up vote method with hashes

NO HOMEWORK TODAY :)