Ruby Closures

for the C and Java minded

Closures

Function / block of code that:

- Can be passed around as a reference (to be called later)
- Can be called outside of it's defining scope
- Keeps the scope it was defined in.

```
message = "Here are some words:"

["foo", "bar", "baz"].each do |word|
  message << " "
  message << word
end

puts message
> Here are some words: foo bar baz
```

```
message = "Here are some words:"

["foo", "bar", "baz"].each do |word|
  message << " "
  message << word
end

puts message
> Here are some words: foo bar baz
```

Closures

Can access variables of the scope they were defined in.

```
def make_greeter
  message = 'Hello world!'

  Proc.new do
    puts message
  end
end
```

```
def make_greeter
  message = 'Hello world!'

  Proc.new do
    puts message
  end
end
```

```
def make_greeter
  message = 'Hello world!'

  Proc.new do
    puts message
  end
end

greeter = make_greeter();
```

```
def make_greeter
  message = 'Hello world!'

  Proc.new do
    puts message
  end
end

greeter = make_greeter();
greeter.call();
> Hello world!
```

```
def make_greeter
  message = 'Hello world!'

  Proc.new do
    puts message
  end
end

greeter = make_greeter();
greeter.call();
> Hello world!

puts message
undefined local variable or method `message'
```

```
class Hello {
  static Runnable makeGreeter() {
    final String message = "Hello world!";
    return new Runnable() {
      public void run() {
        System.out.println(message);
      }
    };
  }

  public static void main(String[] args) {
    Runnable greeter = Hello.makeGreeter();
    greeter.run();
  }
}
```

Closures

Can't access variables in the calling scope.

```
def make_greeter
  Proc.new do
    puts message
  end
end

message = 'Hello world!'
greeter = make_greeter();
greeter.call();
undefined local variable or method `message'
```

Closures

Closures create a new scope, and have their own local variables.

```
def foo
  for i in (1..3)
    x = i
    puts "The value of x is #{x}"
  end
  puts "The final value of x is #{x}"
end

foo
> The value of x is 1
> The value of x is 2
> The value of x is 3
> The final value of x is 3
```

```
def foo
  (1..3).each do |i|
    x = i
    puts "The value of x is #{x}"
  end
  puts "The final value of x is #{x}"
end

foo
> The value of x is 1
> The value of x is 2
> The value of x is 3
undefined local variable or method `x'
```

```
def foo
  x = nil
  (1..3).each do |i|
    x = i
    puts "The value of x is #{x}"
  end
  puts "The final value of x is #{x}"
end

foo
> The value of x is 1
> The value of x is 2
> The value of x is 3
> The final value of x is 3
```

Closures

How to write functions that take blocks?

```
fibonacci_for 5 do |i|
  puts i
end
> 1
> 1
> 2
> 3
> 5
```

```
def fibonacci_for count
  a = 1
  b = 1

  while count > 0
    a, b = b, a + b
    count -= 1
  end

end
```

```
def fibonacci_for count
  a = 1
  b = 1

  while count > 0
    # Do something with `a`
    a, b = b, a + b
    count -= 1
  end

end
```

```
def fibonacci_for count, &block
  a = 1
  b = 1

  while count > 0
    block.call a
    a, b = b, a + b
    count -= 1
  end

end
```

```
def fibonacci_for count
  a = 1
  b = 1

  while count > 0
    yield a
    a, b = b, a + b
    count -= 1
  end

end
```

Closures

Want to investigate further?

- Procs, lambdas, methods
- Passing arguments
- What does `return` do to each of them