1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

arr.select { |n| n.odd? }
```

The local variable `arr` is initialized on line 1 and points to an array object of integers, `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The `#select` method is called on the array object `arr` is pointing to with a `{}` block passed to it as an argument. Each element in the array object is passed into the block in turn and assigned to the block parameter `n`. The `#select` method cares about the truthiness of the block's return value and will return that element if the block's return value evaluates to true. The `#odd?` method is called on the local block variable `n` and will return the boolean true if the integer is odd and false if it is even. All of the integers where the block's return value evaluates as true will be returned in a new array, `[1, 3, 5, 7, 9]`.

This example demonstrates how the `#select` method works. `#select` cares about the truthiness of the return value and will perform selection based off the return value with the selected elements in a new array.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.map do |n|
  n > 1
end
p new_array
```

The local variable `arr` is initialized on line 1 and points to an array object of integers, `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The local variable `new_array` is initialized on line 3 and points to the return value of the `#map` method being called on the array object `arr` is pointing to. The `#map` method is passed a `do..end` block as an argument and each element in the calling collection will be passed into the block in turn and be assigned to the block parameter `n`. the `#>` method evaluates the two integers on the right and left and will return the boolean true if

the integer on the left is greater than the integer on the right. `#map` uses the return value, a boolean in this case, to perform transformation on the calling collection and return a new array with the transformed elements. The `#p` method is called on line 6 with the array object `new_array` is pointing to which will output and return the calling object, `[false, true, true, true, true, true, true, true, true, true]`.

This example demonstrates how the `#map` method works. The return value of the block is used to perform transformation on the calling collection and the transformed elements will be returned in a new array, both collections having the same number of elements.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
{ a: "ant", b: "bear", c: "cat" }.all? do |key, value|
  value.length >= 3
end
```

The `Hash#all?` method is called on a hash of key-value pairs, `{ a: "ant", b: "bear", c: "cat" }`. Each key and value in the calling collection will be passed into the block in turn and be assigned to the block parameters `key` and `value` respectively. The `#all?` method will return a boolean based on the return value of the block. If every iteration of the block returns the boolean true, the `#all?` method will return `true`. The `#length` method is called on the string object `value` is pointing to and will return the character length of the string. The `#>=` method will evaluate the two integers and will return true if the integer on the left is greater than or equal to the integer on the right, false if vice versa. If the return value of `#>=` on each iteration returns true, then the `#all?` method will return true. This method will return `true`.

This example demonstrates how the `#all?` method works. The `#all?` method will return the boolean true if the block's return value for each iteration evaluates as true.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
{ a: "ant", b: "bear", c: "cat" }.each_with_object({}) do |(key, value), hash|
  hash[value] = key
end
```

The `Hash#each_with_object` method is called on a hash of key-value pairs, `{ a: "ant", b: "bear", c: "cat" }`, with an empty hash passed in and a `do..end` passed in as an argument. Each key and value pair will be passed into the block in turn and assigned to the first block parameter `(key, value)`, respectively and the empty hash will be passed into the second block parameter `hash`. The `[]` method will initialize a key in the empty hash as the string object local block variable `value` is pointing to and it's value as the symbol local block variable `key` is pointing to which will be returned by the method, `{"ant"=>:a, "bear"=>:b, "cat"=>:c}`.

This example demonstrates how the `#each_with_object` method works. Each element in the calling object is passed into the block being assigned to the first block parameter and the object which is passed in as the argument to the method will be passed into the block being assigned to the second block parameter. The method will return the newly populated object which was passed in as the argument to the method.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

   What are a, b, and c? What if the last line was `c = a.uniq!`?

```
a = [1, 2, 3, 3]
b = a
c = a.uniq
```

The local variable `a` is initialized on line 1 and points to an array object of integers, `[1, 2, 3, 3]`. The local variable `b` is initialized on line 2 and points to the same array object `a` is pointing to. The local variable `c` is initialized on line 3 and points to the return value of the `#uniq` method being called on the array object `a` is pointing to which will return a new array with the duplicate elements deleted, `[1, 2, 3]`. `a` is pointing to the array object `[1, 2, 3, 3]`. `b` is pointing to the same array object as `a`, `[1, 2, 3, 3]` and `c` is pointing to array object `[1, 2, 3]`. If line 3 was `c = a.uniq!`, all three variables would be pointing to the array `[1, 2, 3]`. This is because `#uniq!` is a destructive method which will mutate the calling object where `#uniq` is not a destructive method.

This example demonstrates variables as pointers. When variables are initialized, they point to a space in memory. When we reassign variables, we are changing the space in memory that the variable is pointing to. If two variables are pointing to the same string object and one gets reassigned, this does not affect the other variable.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

   What values do `s` and `t` have? Why?

```ruby
def fix(value)
 value[1] = 'x'
 value
end

s = 'abc'
t = fix(s)
```

The local variable `s` is initialized on line 6 and points to string object `'abc'`. The local variable `t` is initialized on line 7 and points to the return value of the method `fix` with the string object `s` is pointing to passed in as an argument. The method definition for `fix` are on lines 1-4. The string object `s` is pointing to will be assigned to the method parameter `value`. The destructive `#[]=` method will replace the letter at the index location `1` to the letter `'x'`, `'axc'` which the method will implicitly return. `s` and `t` are both pointing to the string object `'axc'`.

```
This example demonstrates variables as pointers.  When variables are
initialized, they point to a space in memory.  The destructive `#[]=` action
within the method will affect the calling object.
```

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
def a_method(string)
  string << ' world'
end

a = 'hello'
a_method(a)

p a
```

The local variable `a` is initialized on line 5 and points to string object `'hello'`. The `a_method` is called on line 6 with the string object `a` is pointing to passed in as an argument. The method definition for `a_method` are on lines 1-3. The string object `a` is pointing to will be assigned to the method parameter `string`. The destructive `#<<` method will append the string object `'world'` to the calling object and the method will implicitly return it. The `#p` method is called on line 8 with the string object `a` is pointing to passed in as an argument which will output and return `'hello world'`.

This example demonstrates variables as pointers. When variables are initialized, they point to a space in memory. The destructive `#<<` action within the method will affect the calling object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

   What values do `s` and `t` have? Why?

```
def fix(value)
  value = value.upcase!
  value.concat('!')
end

s = 'hello'
t = fix(s)
```

The local variable `s` is initialized on line 6 and points to the string object `'hello'`. The local variable `t` is initialized on line 7 and points to the return value of the `fix` method being called with the string object `a` is pointing to passed in as an argument. The string object `a` is pointing to will be assigned to the method parameter `value`. The local method variable `value` will be reassigned to the return value of the destructive `#upcase!` method being called on the string object `value` is pointing to which will return `'Hello'`. The destructive `#concat` method is called on the string object `value` is pointing to which will return `'Hello!'` which the method will implicitly return. `s` and `t` are both pointing to the same mutated string object `'Hello!'`.

This example demonstrates variables as pointers. When variables are initialized, they point to a space in memory. Both the `#upcase!` and `#concat` methods are destructive actions which will affect the calling object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = “Hello”
b = a
a = “Goodbye”
puts a
puts b
```

The local variable `a` is initialized on line 1 and points to string object `'Hello'`. The local variable `b` is initialized on line 2 and points to the same string object `a` is pointing to `'Hello'`. The local variable `a` is reassigned to point to the string object `'Goodbye'`. The `#puts` method is called on line 4 with the string object `a` is pointing to passed in as an argument which will output it, `'Goodbye'` and return `nil`. The `#puts` method is called on line 5 with the string object `b` is pointing to passed in as an argument which will output it, `'Hello'` and return `nil`.

This example demonstrates variables as pointers. When variables are initialized, they point to a space in memory and are not bound to that object. Reassignment will only affect the variable it is being performed on and not any other variable that is pointing to the same object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = "hello"

[1, 2, 3].map { |num| a }
```

The local variable `a` is initialized on line 1 and points to string object `'hello'`. The `#map` method is called on the array object of integers with a `#{}` block passed to it as an argument. Each element in the calling array object is passed into the `{}` block once in turn and gets assigned to the block parameter `num`. Map takes the return value of the block to perform transformation which then each transformed element will be input into a new array which will be returned by the method. This block of code will output nothing and return `['hello', 'hello', 'hello']`.

This example demonstrates how the `#map` method works. `#map` will take the return value of the block passed to it to perform transformation on the elements in the calling object and return a new array with the transformed elements. `#map` will return a new array with the same number of elements as in the calling object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
[1, 2, 3].each do |num|
  puts num
end
```

The `#each` method is called on the array object of integers with a `do..end` block passed to it as an argument. Each integer object in the calling array will be passed into the block in turn and assigned to the block parameter `num`. Within the block, the `#puts` method is called with the integer object local block variable is pointing to in turn and will output it and return `nil`. The `#each` method does not use the return value of the block and will return the calling object `[1, 2, 3]`.

This example will output `1`, `2`, `3`, on separate lines and return the calling object `[1, 2, 3]`. This example demonstrates how the `#each` method works. The `#each` method does not care about the return value of the block and will return the calling object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

incremented = arr.map do |n|
                n + 1
              end
p incremented
```

The local variable `arr` is initialized on line 1 and points to an array object of integers, `1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The local variable `incremented` is initialized on line 3 and points to the return value of the `#map` method being called on the array object `arr` is pointing to. The `#map` method is passed a `do..end` block as an argument. Each element in the calling array will be passed into the block in turn and be assigned to the block parameter `n`. The `#map` method takes the return value of the block to perform transformation. The `#+` method is called on the integer object `n` is pointing to and will increment it by 1. The incremented integers are then returned by the `#map` method in a new array. The `#p` method is called on line 6 with the array object `incremented` is pointing to passed in as an argument which will output and return the calling object, `[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`.

This example demonstrates how the `#map` method works. The `#map` method takes the return value of the block to perform transformation on the elements in the calling object and return a new array with the transformed elements. `#map` will return a new array with the same number of elements as in the calling object.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 4

loop do
  a = 5
  b = 3

  break
end

puts a
puts b
```

The local variable `a` is initialized on line 1 and points to integer object `4`. The `#loop` method is called on line 3 with a `do..end` block passed in as an argument. The local variable `a` is reassigned to point to integer object `5`. The local block variable `b` is initialized and points to integer object `3`. The `break` breaks us out of the loop. The `#puts` method is called on line 11 and the integer object `a` is pointing to is passed in as an argument which will output it, `5` and return `nil`. Line 12 will cause an error. This is because the variable `b` was initialized within the block so is not accessible outside the block.

This example demonstrates block scoping rules. Variables initialized in an outer scope are accessible within a block but variables initialized within a block are not accessible outside the block. Since variable `b` was initialized within the block, the `#puts` method does not have access to it in the outer scope.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
a = 'Bob'

5.times do |x|
  a = 'Bill'
end

p a
```

The local variable `a` is initialized on line 1 and points to the string object `'Bob'`. The `#times` method is called on the integer `5` with a `do..end` block passed to it as an argument. The `#times` method will invoke the block `5` times and each time we invoke the block, we are passing it to an incrementing value, starting from `0`. Each incrementing value will be passed into the block in turn and be assigned to the block parameter `x`. The variable `a` is reassigned to point to the string object `'Bill'`. The `#times` method will output nothing and return `5`. The `#p` method is called on line 7 with the string object `a` is pointing to passed in as an argument which will output and return the calling object, `'Bill'`.

This example demonstrates variables as pointers. The variable `a` which was initialized in the outer scope is accessible within the block so the reassignment will affect the variable in the outer scope.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
def increment(x)
  x << 'b'
end

y = 'a'
increment(y)

puts y
```

The local variable `y` is initialized on line 5 and points to string object `'a'`. The `increment` method is called on line 6 with the string object `y` is pointing to passed in as an argument. The method definition for `increment` are on lines 1-3. The string object `y` is pointing to will be assigned to the method parameter `x`. The destructive `#<<` method will append the string object `'b'` to the calling object `'a'`. The mutated string object will be implicitly returned by

the method. The `#puts` method is called on line 8 with the string object `y` is pointing to passed in as an argument which will output it, `'ab'`, and return `nil`.

This example demonstrates variables as pointers. When variables are initialized, they point to a space in memory and are not bound to that object. The destructive `#<<` method will mutate the calling object by appending the string object `'b'` to the calling object and returning it.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr1 = ["a", "b", "c"]
arr2 = arr1.dup
arr2.map! do |char|
  char.upcase
end

puts arr1
puts arr2
```

The local variable `arr1` is initialized on line 1 and points to an array object of integers, `['a', 'b', 'c']`. The local variable `arr2` is initialized on line 2 and point to the return value of the `#dup` method being called on the array object `arr1` is pointing to which will return a copy of the calling object. The destructive `#map!` method is called on the array object `arr2` is pointing to with a `do..end` block passed in as an argument. Each element in the calling array is passed into the block in turn and is assigned to the block parameter `char`. The `#map!` method takes the return value of the block to transform the elements in the calling object. The `#upcase` method is called on the string object local block variable `char` is pointing to in turn and will mutate the calling object. The `#puts` method is called on line 7 with the array object `arr1` is pointing to passed in as an argument which will output it, `['a', 'b', 'c']` and return `nil`. The `#puts` method is called on line 8 with the array object `arr2` is pointing to passed in as an argument which will output it, `['A', 'B', 'C']` and return `nil`.

This example demonstrates how the destructive `#map!` method works. The return value of the block will be used to perform transformation. The non destructive `#map` method will return a new array with the transformed elements, but since the `#map!` method is destructive to the calling object, the calling object will be mutated.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
arr = [1, 2, 3, 4]

counter = 0
sum = 0

loop do
  sum += arr[counter]
  counter += 1
  break if counter == arr.size
end

puts "Your total is #{sum}"
```

The local variable `arr` is initialized on line 1 and points to an array object of integers, `[1, 2, 3, 4]`. The local variable `counter` is initialized on line 3 and points to integer object `0`. The local variable `sum` is initialized on line 4 and points to integer object `0`. The `#loop` method is called on line 6 with a `do..end` block passed to it as an argument. Within the block, `sum` is reassigned to the return value of the integer object `sum` is pointing to incremented by the integer object at the indexed location of the current value of `counter` in the `arr` array. The `counter` is incremented by `1` on line 8. The `break` will break out of the loop if the condition `counter == arr.size` is met. The `#size` method is called on the array object `arr` is pointing to which will return the integer value of the number of elements in the array, `4`. The `#puts` method is called on line 12 with a string object passed in as an argument which will output it, `"Your total is 10'` using string interpolation.

This example demonstrates variables as pointers. The variables initialized in the outer scope are accessible within the block which allow reassignment of the variable to happen.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.select do |n|
  n + 1
end
p new_array
```

The local variable `arr` is initialized on line 1 and points to an array object of integers, `[1, 2, 3, 4, 5, 6, 7, 8, 9,10]`. The local variable `new_array` is initialized on line 3 and points to the return value of the `#select` method being called on the array object `arr` is pointing to with a `do..end` block passed to it as an argument. Each element in the calling array will be passed into the block in turn and be assigned to the block parameter `n`. The `#select` method cares about the truthiness of the return value of the block. The local block variable `n` is incremented by `1` which will evaluate as true for each iteration. Since the return value of the block evaluates as true for each iteration, each element in the calling array will be selected and returned in a new array, `[1, 2, 3, 4, 5, 6, 7, 8, 9,10]`.

This example demonstrates how the `#select` method works. `#select` cares about the truthiness of the return value of the block and will perform selection based on it. Since everything in Ruby is truthy except for the boolean `false` and `nil`, the block's return value will evaluate as true for each iteration.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
[1, 2, 3].any? do |num|
  num > 2
end
```

The `#any?` method is called on the array object of integers, `[1, 2, 3]` with a `do..end` block passed to it as an argument. Each element in the called object will be passed into the block in turn and be assigned to the block parameter `num`. The `#any?` method will return `true` if on any iteration, the block's return value returns `true`. The `#>` method will evaluate the two integers and return true if the integer on the left is greater than the integer on the right, false if vice versa. Since on the second iteration, the block's return value is `true`, the `#any?` method will return `true`.

This example demonstrates how the `#any?` method works. The block's return value is evaluated on each iteration and if the return value for any iteration evaluates as true, the method will return `true`.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

   What values do `s` and `t` have? Why?

```ruby
def fix(value)
  value.upcase!
  value += '!'
  value
end

s = 'hello'
t = fix(s)
```

The local variable `s` is initialized on line 7 and points to the string object, `'hello'`. The local variable `t` is initialized on line 8 and points to the return value of the `fix` method being called with the string object `s` is pointing to passed in as an argument. The method definition for `fix` are on lines 1-5. The string object `s` is pointing to will be assigned to the method parameter `value`. The destructive `#upcase!` method is called on the local method variable `value` and will mutate the calling object to `'Hello'`. The local method variable `value` is reassigned to the return value of the string object `value` is pointing to with the string object `'!'` appended to it, `'Hello!` which will be implicitly returned by the method. The local variable `s` is pointing to the string object `'Hello'` and local variable `t` is pointing to the string object `'Hello!'`.

This example demonstrates mutability and reassignment. Some methods in Ruby are destructive and will mutate the calling object. Reassignment is a non destructive action. The destructive `#upcase!` method mutates the calling object that `s` is pointing to. The reassignment on line 3 changes the string object that the method variable `value` is pointing to and not variable `s`.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
a = 4
b = 2

loop do
  c = 3
  a = c
  break
end

puts a
puts b
```

The local variable `a` is initialized on line 1 and points to integer object `4`. The local variable `b` is initialized on line 2 and points to integer object `2`. The `#loop` method is called on line 4 with a `do..end` block passed to it as an argument. The local block variable is initialized and points to integer object `3`. The local variable `a` is reassigned to point the integer object `c` is pointing to , `3`. The `break` breaks us out of the loop. The `#puts` method is called on line 10 with the integer object `a` is pointing to passed in as an argument which will output it, `3` and return `nil`. The `#puts` method is called on line 11 with the integer object `b` is pointing to passed in as an argument which will output it, `2` and return `nil`.

This example demonstrates block scoping rules. Variables initialized in an outer scope are accessible within a block. The local variable `a` is accessible within the block and is reassigned.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
a = %w(a b c)
a[1] = '-'
p a
```

The local variable `a` is initialized on line 1 and points to an array object of string objects, `["a", "b", "c"]`. The destructive `#[]=` method will mutate the element at the indexed position of `1` to `'_'`. The `#p` method is called on line 3 and the array object `a` is pointing to will be passed in as an argument which will output and return the calling object, `["a", "-", "c"]`.

This example demonstrates mutability. The variable `arr` is pointing to the array object, but the elements within the array are objects themselves which can be mutated. Since `#[]=` is a destructive method, the object within the array will be mutated.

1. What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def add_name(arr, name)
  arr = arr + [name]
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names
```

The local variable `names` is initialized on line 5 and points to an array object of strings, `['bob', 'kim']`. The `add_name` method is called on line 6 with the array object `names` is pointing to passed in as the first argument and the string object `'jim'` passed in as the second argument. The method definition for `add_name` are on line 1-3. The first and second arguments will be assigned to the method parameters `arr` and `name` , respectively. The local method variable `arr` will be reassigned to the return value of `#+` method being called on the array object `arr` is pointing to with the string object `name` is pointing to passed in as an argument which will return `['bob', 'kim', 'jim']`. This will be returned by the method. The `#puts` method is called on line 7 and the array object `arr` is pointing to will be passed in as an argument which will output it, `['bob', 'kim']` and return `nil`.

This example demonstrates pass by value. Since the method `add_name` is a non destructive method, a copy of the array object `arr` is pointing to will be passed in as the argument.