##Local Variables

1. The last line outputs 2 because the variable `str` (initialized at line 1, outside of the block) was set to 2 in the `loop` block at line 4.

2. The code in this example raises an exception because the variable `str` was only available to the local scope of the `loop` block. Once the program control left the `loop` block in lines 1-4, the `str` variable fell out of scope. (We know that the `str` variable was not set earlier in the program because the code snippet starts at line 1.)

3. You cannot say with certainty if the code in the snippet will run because we cannot know whether the variable `str` was initialized (or what it may have been initiated to) earlier in the program.

4. The code here raises an exception because the `str` variable is outside of the scope of the method. Unlike in a block, a method does not have awareness of variables outside of its scope unless they are explicitly passed in to the method as an argument.

5. The last line outputs "hello" because the `str` argument to `puts` is the `str` in the outer scope (set at line 1.) The `str` set to 'world' in line 4 is within the body of the `a\_method` and is a variable local to the method. That local variable is out of scope outside of the method. The two `str` variables are not the same variable -- they simply have the same name.

6. In line 2, b is set to the same string as `a`. At line 3, `a` is reassigned to a new string, "hi", while `b` is still set to the original string, "hello". At line 4, the string at `a` is modified with the shovel operator to a new value, "hi world". Yet `b` is still set to the original string "hello" from line 2. The shovel operator only modified the current value of `a`, not the old value of `a`.

7. There are two objects and four variables in the code.

Object 1, Variable 1: At line 1, the "hello" object is created and set to the first variable, `a`.

Object 2, Variable 2: At line 2, the "world" object is created and set to the second variable, `b`.

Variable 3: At line 4, the third variable, `c` is set to the value at `a`.

Variable 4: At line 5, the fourth variable, `d` is set to the value at `b`.

Lines 6 and 7 change the values assigned to `b` and `a` but they do not create any new variables or objects.

##Mutating Method Arguments

1. The last line in the code prints "hello" because the method called `change` doesn't actually change the value of the argument passed in. Inside the method, the value of `greeting` is assigned to variable in the local scope of the method and then the method ends without modifying the argument. This method demonstrates a non-destructive method that does not modify arguments passed in.

2. The last line in this code, however, demonstrates that this `change` method is destructive and changes the argument passed in. Within the `change` method, the argument is modified using the shovel (<<) operator, which is destructive. The shovel operator actually modifies the value at `greeting` to be "hello world."

3. The last line will output "hello". By adding `param = "hi"`, the method now creates a new local variable called `param` (because reassignment in a method creates a local variable, sets it to "hi" at line 2 and then modifies the local variable to be "hi world" with the shovel operator at line 3. The changes made to `param` are lost to the inner scope of the method.

4. In this code, similarly to #3 above, the `change` method immediately reassigns `param` to a new value. The shovel operator on line 3 (and later on line 5) is modifying the local variable `param`, not the value held by the original argument passed in, `greeting`. When we leave the scope of the method, the outer-scope variable still holds isn't original value, "hello".

##Working with Collections

1. Array#map creates a new array and carries out or executes the code held in the associated block for each element in the array. Array#map assigns the returned value from the code in the block to each corresponding element of the new array. Array#map then returns the newly created array of return values. If there's no code in the block to execute, Array#map will return an Enumerator of the array instead.

There is a destructive version of Array#map, called Array#map! which, instead of creating and returning a new array to store the returned values from the code block, will modify each element of the original array.

2. Array#select creates a new array and then pushes to the newly created array any values of the original array that meet the conditions described in the code block. Array#select then returns the newly created array of items that meet that condition. If no code block is given, then Array#select returns an Enumerator.

There is a destructive version of Array#select, called Array#select! which, instead of creating a returning a new array to store the elements of the original array that meet the condition in the block, will delete items from the original array that don't meet the conditions in the block. Array#select! then returns `self’. If all elements in the array meet the conditions and so no elements are deleted, then Array#select! will return nil.

3. Enumerable#reduce is a little like Array#map in that it steps through the elements in an enumerable and executes the code in the given block against the element. Unlike `map`, however, `reduce` then adds the results from that code all together in a new variable. Enumerable#reduce then returns the total of all `reduce` results.

4. The two different `map` lines will return an array containing `[2, 3, 4]`. Each steps through an array containing `[1, 2, 3]`. The first adds 1 to n and pushes that value to the newly created results array. The second `map` reassigns the local block variable `n` to n + 1. I prefer the first because it has less overhead and could have a noticeable improvement in performance for large arrays.

5. The code in the `map` block is evaluating whether `n` is greater than 2. `map` returns an array with the return values of the block -- in this case, it would be [false, false, true]. `select` would instead return [3], which is all of the elements than meet the condition.

6. Now the last value in the block is the return value of `puts`, so `map` will return an array of [`nil`, `nil`, `nil`] because puts will output the argument (`n`) in this case, but only returns `nil`.

7. Array#select returns an array with each element that meets the conditional in the block - but there's no conditional in this block, so each element would meet it. If the coder wanted an array of the elements with 2 added to each, s/he should have used:

`arr = [1, 2, 3].map do |n|`

` n + 2`

`end`

``

`p arr`

8. The value of `arr` will be an empty array because none of the elements is nil (and `puts` returns `nil`).