# W3D2

## References & Scope

---

## Today's Learning Objectives

- Describe the differences between variables/references and actual Ruby objects

- Identify whether an operation mutates an object or reassigns a variable reference

- Given a code snippet, determine which variables are in scope in each section

- With regards to scope, explain the different implications of opening up a block vs a method definition

- Properly initialize a `Hash` and `Array` with default values

---

## What we WON'T be covering

- Memory allocation/management in Ruby (beyond basics of Object ID)

- Garbage collection

---

## References in Ruby

---

### Variables

+ Ruby variables hold \*references\* (otherwise known as \*pointers\*) to objects stored in memory

+ `=` assignment operator

+ \_assigns\_ the variable pointer

+ doesn't change or \*mutate\* the object stored in memory

---

### Mutability

+ Mutable: state can be modified after it is created

+ Immutable: state cannot be modified after it is created

| Mutable | Immutable |

|----------|-------------|

| `String` | `Integer` |

| `Array` | `Float` |

| `Hash` | `Symbol` |

---

### Reference Diagram + Demo

---

### What's the difference

\* `<<`

\* `concat`

\* `+=`

---

## 5 minute break

---

# Scope

---

## Different kinds of variables

\* \*\*Local variables\*\*

\* Global Variables

\* Instance variables

\* Class Variables

---

### Block Scoping (of local variables)

\* Inside of a `do..end` block, you have access to all variables declared previously (higher) in your code and at the same or an outer-level scope.

\* Variables \*declared\* within a block, are not accessible to the outer scope

\* However, if a variable is declared in outer-level scope, \*changes\* made to variable in inner-level scope persist when back at outer-level.

---

## Examples

---

```rb

x = 2

3.times do

x \*= 2

end

puts x

```

---

```rb

3.times do

x = 2

end

puts x

```

---

### Scope "Gates"

Parts of Ruby code in which an \*entirely\* new scope is created and all local variables previously defined are no longer accessible (i.e. you do not have access to local variables defined in outer scopes). The three main scope gates are:

1. Method definitions

2. Class definitions

3. Module definitions

---

## Examples

---

```rb

x = 10

def some\_method

puts x

end

some\_method

```

---

## Parameters vs Arguments

\* \*\*Parameter\*\* - variable in a method \_definition\_

\* \*\*Argument\*\* - variable method is \_invoked\_ with

---

## Passing arguments to methods

\* In the scope of a method, the parameter is a copy of the argument that the method was invoked with - both are references

\* In essence another reference is created, that points to the same spot in memory as the argument passed in

\* Therefore if we \_reassign\_ the parameter within the method, the argument that the method was invoked with is unaffected

\* But if we \_mutate\_ the parameter within the method, this affects the argument the method was invoked with (since it is pointing to the same object)

---

### What do you think this code will output?

```rb

def inc(num)

num += 1

end

a = 1

inc(a)

puts a

```

---

### How about this?

```rb

def add\_square(arr, num)

arr << (num \*\* 2)

end

squares = [1, 4, 9]

add\_square(squares, 4)

p squares

```

---

## [Kahoot!](https://play.kahoot.it/v2/?quizId=5ad379b2-90e6-405f-bb15-e2b854a4c9dc)

---

## 5 minute break

---

## Array/Hash Defaults

---

## Making Array of Arrays

\* Naive Approach: `Array.new(3, [])`

\* Correct Approach(es): `Array.new(3) { [] }` || `Array.new(3) { Array.new }`

---

## Making Hash with Array default

\* Naive Approach 1: `Hash.new([])`: The default could be mutated!

\* Naive Approach 2: `Hash.new { [] }`: does not store keys into the hash!

\* Correct Approach: `Hash.new { |h, k| h[k] = [] }`

---

## Creating a Counter Hash

\* `Hash.new(0)`: works fine because we can't mutate Integers

---

## [Kahoot!](https://play.kahoot.it/v2/?quizId=14eb8cff-210c-4522-aa4c-5bb4aa9ee86b)

---

# Thank You!