Arrays

Link to Ed Lesson

A link to the lesson can be found here

What is an Array?

An *array* is a collection of items. Instead of creating multiple variables to store data, we can group **similar kinds of data** in an *array*. Example 1 shows a poor approach to storing data by storing similar data in multiple variables while example 2 illustrates a better alternative by using an array.

1.
```
    Educator_1 = "Isaac"
    Educator_2 = "Ben"
    Educator_3 = "Sam"
    Educator_4 = "Will"
```

2.
```
    Educators = ["Isaac" , "Ben", "Sam", "Will"]
```

Syntax for Array data type

The syntax for an *array* is a **square bracket** followed by **comma seperated values** and finishes with a **closing sqaure bracket** (as above).

Array can hold multiple data types

- *Arrays* are NOT limited to *strings*. *Arrays* can also store *integers* or even other *arrays*.

```
  integer_array = [ 0, 1 , 2 , 3 , 4]
```

- Ruby allows for **mutlple data types** to be stored within an *array*. Therefore, we are NOT LIMITED to just **one data type** when using *arrays*.

```
  multiple_data_array = ["index_0", 1 , true, false, ["yellow",
"red"]]
```

Syntax for accessing array elements

```
variable_name[index]
```

- Array index always starts at 0.
- If a value doesn't exist at a particular position, the array will return NILL.
- Elements in an array can be accessed using negative indexing.

Array methods

- *Arrays* have a number of **predefined methods** to simply access and **manipulate array elements**.

  ```
  #common array methods

  #returns total count of elements in array
  my_array.length

  #combines the array elements into a single element
  my_array.join

  #checks if item exists in array returns boolean value
  my_array.include?

  #deletes specified item from array via item name
  my_array.delete

  #deletes specified item from array via item index
  my_arrary.delete_at
  ```

Array Mutations

- Whenever we look at ANY VARIABLE, whether it's a *string* or an *integer*, when you copy that particular variable and we change the original value, the copied value will NOT change. For example:

  ```
  #creates a variable with the value 10
  num = 10

  #creates a new variable with the value of the num variable (10)
  copy_num = num #assigns num variable to a new variable
  ```

```
  #changing the num variable to new value will not effect the copy_num
variable
  num = 20

  copy_num
  => 10
```

- This is because the num variable and copy_num variable are pointing to **two different values** in the **memory location**.
- However with an *array*, when we have two different variables created and when make a copy they STILL point to the **same memory location**. Therefore, it is a COPY of the same memory location.
- In this manner, if we CHANGE the value of the original array, since they are pointing to the SAME memory location, the copied array will ALSO BE effected by the changes made to the **original array**.
- This happens because **both are pointing to the same memory location** and because arrays are passed by reference (which is the memory location) and are NOT passed by the **value**.
- To overcome this problem, we need to explicitly CLONE the array by using the **clone method**.
- Therefore, whenever we are making a copy of an array we need to make a clone of it. See example below:

```
  #sets an array
  names = ["isaac", "kaylee", "seb" "fred"]

  #sets a new array which clones the names array
  copy_names = names.clone

  #updates index 0 of names array
  names[0] = "leonard"

  #displays the updated names array
  puts names
  => leonard
  => kaylee
  => seb
  => fred

  #However, the coype_names will not be effected
  puts copy_names
  => isaac
  => kaylee
  => seb
  => fred
```

- In other words, if you make a copy of an array WITHOUT using the clone method, the copied variable WILL change its value **even if we don't explicitly change its value**.
- However, if we use the clone method, modifications will ONLY effect the original array. **This is why the clone method is so useful**.

Error Handling

- While applying methods on array elements, **a short circuit logic** can be used to CHECK if the value exist **before applying a method on it**.

1.
   ```
   names = ["isaac", "kaylee", "seb", "fred"]

   puts names[0].capitalize #Isaac
   puts names[1].capitalize #Kaylee
   puts names[2].capitalize #Seb
   puts names[3].capitalize #Fred

   puts names[4].capitalize #ERROR!
   ```

2.
   ```
    puts names[0].capitalize #Isaac
    puts names[1].capitalize #Kaylee
    puts names[2].capitalize #Seb
    puts names[3].capitalize #Fred

    puts names[4] && names.capitalize #NIL
   ```

- The above example solves this problem by using a logical && operator where it first checks if names[4] is true and since it returns false, the names.capitalize portion of the code will not be executed and the return will be NILL.
- Therefore, it is wise to include the && logical operator to check if the element is in the array before applying array methods.

Multi-dimensional Arrays

```
#normal array
names = ["isaac", "kaylee", "seb", "fred"]

#multi-dimensional array
odd_even = [[1,3,5,7],[2,4,6,8]]

#access the first list nested inside the array
```

```
    print odd_even[0]
    => [1,3,5,7]

    #access the second list nested inside the array
    print odd_even[1]
    => [2,4,6,8]
```

- In order to access elements within the first array, you need to add a second set of square brackets and add the index of the nested array element.

  ```
    #multi-dimensional array
    odd_even = [[1,3,5,7],[2,4,6,8]]

    #accessing the second element inside the first list nested inside the array.
    print odd_even[0][1]
    => 3

    #accessing the second element inside the second list nested inside the array.
    print odd_even[1][1]
    => 4
  ```