GENERAL ASSEMBLY

MOBILE DEVELOPMENT

SWIFT DATA STRUCTURES

William Martin
Head of Product, Floored

LEARNING OBJECTIVES

- Compare and contrast Tuples, Arrays, and Dictionaries.
- Create and modify Tuples.
- Deploy Tuples to return multiple values from a function.
- Create, modify, and iterate over Arrays.
- Identify and use Array methods and properties.
- Create, modify, and iterate over Dictionaries.
- Identify and use Dictionary methods and properties.

REVIEW OF IBOUTLETS AND IBACTIONS

WHAT ARE DATA STRUCTURES?

WHAT IS A DATA STRUCTURE?

- A data structure is a type that manages potentially large amounts of data.
- But what are data?

WHAT ARE DATA?

- Data are pieces of information devoid of meaning.
- e.g. What do these numbers mean?
  - 83, 80, 82, 87, 86, 88, 78, 87, 88, 89, 85, 85, 88, 81, 83, 87, 90, 88, 95, 88, 96, 94, 93, 92, 89, 92, 86, 88, 97, 92, 89

WHAT ARE DATA?

- Data are pieces of information devoid of meaning.
- e.g. What do these numbers mean?
  - 83, 80, 82, 87, 86, 88, 78, 87, 88, 89, 85, 85, 88, 81, 83, 87, 90, 88, 95, 88, 96, 94, 93, 92, 89, 92, 86, 88, 97, 92, 89
- Hint: We don't know.

WHAT ARE DATA?

- Data are pieces of information devoid of meaning.
- e.g. What do these numbers mean?
  - 83, 80, 82, 87, 86, 88, 78, 87, 88, 89, 85, 85, 88, 81, 83, 87, 90, 88, 95, 88, 96, 94, 93, 92, 89, 92, 86, 88, 97, 92, 89
- Hint: We don’t know.
- But when I tell you that they’re the record high temperatures for NYC in May 2015 collected in Central Park, then they *mean* something. They’re promoted to *information*.

DATA STRUCTURES

TUPLES

WHAT IS A TUPLE?

- An tuple is a type that represents a small list of things called "elements".
- Tuples bundle multiple values of *potentially different types* into a single value. We call this "heterogeneous typing."

SYNTAX

Tuple literal syntax is a comma-separated list between parentheses:

```
let constants = (3.14, 2.718, 1.618)
```

Access tuple elements by using dot-index syntax:

```
constants.0

constants.1

constants.2
```

SYNTAX

Tuple elements can also be given names, just like parameters in a function definition.

```
let constants = (pi:3.14, e:2.718, phi:1.618)
```

Access tuple elements by using those names (or dot-index syntax as well):

```
constants.pi
constants.e
constants.phi
```

MULTIPLE TYPES

Tuple types are declared using a similar comma-separated, parenthetical syntax, but using types instead of values (assume Gender is an enum with cases Male and Female):

```
let personData : (name:String, age:Int, gender:Gender) =
    (name:"Emily", age:32, gender:.Female)
```

You can also use class names here, since classes are themselves types.

BUNDLING VALUES

Tuples are often used to return multiple values from a function:

```swift
enum Decade {
    case Twenties
    case Thirties
}

func getAgeRange(decade:Decade) -> (Int, Int) {
    switch decade {
    case .Twenties:
        return (20, 29)
    case .Thirties:
        return (30, 39)
    }
}
```

SPECIAL RULES

There is no such thing as a one-element tuple. It's just a single value of its own type.

```
let single = (3)  // Ooops! Just an Int, not a Tuple of Int.
```

There is an empty Tuple, equivalent to Void, which we've already seen deployed to indicate a function does not return a value (i.e. a "Void function").

```
let empty = ()
```

DATA STRUCTURES

ARRAYS

WHAT IS AN ARRAY?

- An array is a type that represents a list of things called "elements", typically all of the same type.
- We call this "homogeneous typing."
- Thus we say "Array of Ints" or "Array of Floats" or "Array of Dogs". Arrays can indeed contain instances of classes, even ones we write.
- Arrays can also be empty (i.e. contain no elements), but the elements are still typed!
- *It's used to reference a lot of data points in an ordered way.*

HOW DO YOU DECLARE AN ARRAY TYPE?

- Syntax for declaring the type uses brackets and another Swift type inbetween them.
- Here's an explicitly typed variable containing an Array of Strings.
  - `var words : [String] = ["hello", "doctor"]`

HOW DO YOU DECLARE AN ARRAY TYPE?

- Syntax for declaring the type uses brackets and another Swift type inbetween them.
- Here's an explicitly typed variable containing an Array of Strings.
  - `var words : [String] = ["hello", "doctor"]`
- Here's an implicitly typed constant containing an Array of Doubles
  - `let numbers = [3.14, 2.718]`

HOW DO YOU DECLARE AN ARRAY TYPE?

- Syntax for declaring the type uses brackets and another Swift type inbetween them.
- Here's an explicitly typed variable containing an Array of Strings.
  - `var words : [String] = ["hello", "doctor"]`
- Here's an implicitly typed constant containing an Array of Doubles
  - `let numbers = [3.14, 2.718]`
- Here's an explicitly typed variable with an empty Array of Strings.
  - `var words : [String] = []`

HOW DO YOU DECLARE AN ARRAY TYPE?

- Syntax for declaring the type uses brackets and another Swift type inbetween them.
- Here's an explicitly typed variable containing an Array of Strings.
  - `var words : [String] = ["hello", "doctor"]`
- Here's an implicitly typed constant containing an Array of Doubles
  - `let numbers = [3.14, 2.718]`
- Here's an explicitly typed variable with an empty Array of Strings.
  - `var words : [String] = []`
- Here's an implicitly typed variable with an empty Array of Doubles.
  - `var numbers = [Double]()`
- *Note how the type is instantiated in the last one with (), just like classes!*

ARRAY INDICES

Each element has an Integer **index** that we use to refer to the element. When we do say we're "indexing into the Array."

```
let words = ["hello", "doctor", "name", "continue", "yesterday", "tomorrow"]
```

0	1	2	3	4	5
"hello"	"doctor"	"name"	"continue"	"yesterday"	"tomorrow"

ARRAY INDICES

Each element has an Integer **index** that we use to refer to the element. When we do say we're "indexing into the Array."

```
let words = ["hello", "doctor", "name", "continue", "yesterday", "tomorrow"]
```

We retrieve the first element like this, using brackets containing the Integer index. This is called "subscript notation":

```
let first = words[0]
```

And the second like this. (We say, "words sub one."):

```
let second = words[1]
```

Quick, what type are <u>words</u>, <u>first</u>, and <u>second</u> containing?

ARRAY INDICES

- Arrays are an *ordered* data structure, reflected by the fact that they are indexed by increasing Integers.
- This means there's definitely a notion of a first and last element, a next and previous element to a given one.
- Also, we're *guaranteed an order* if the Array changes (i.e. has elements removed, changed, or added), but we have to *be specific about how the Array is changed.*
- It also means we can use loops to step through the Array automatically (i.e. "iterate over the Array") in that order (beginning to end, end to beginning).

ARRAYS AND LOOPS

Iterating over an Array with a for loop:

```
let words = ["hello", "doctor", "name", "continue", "yesterday", "tomorrow"]
for word in words {
    // This loops six times, one for each element of the Array "words".
    // A constant "word" is available in the body of the loop.
    // "word" will be of type String and contain an Array element.
    // The loop proceeds in order, from index 0 to the end.
}
```

ARRAYS AND LOOPS

Let's pretend we have a special view in our app that shows a list of words:

```
let words = ["hello", "doctor", "name", "continue", "yesterday", "tomorrow"]
for word in words {
    viewShowingListOfWords.addToList(word)
}
```

ARRAYS AND LOOPS

Sometimes, when iterating, we also need the element's index. Here's how to do that. Note how we're using a tuple to extract two values from the loop instead of just one:

```
for (index, element) in words.enumerate() {
    // Loops three times...
    // index is 0, 1, 2, 3, 4, 5.
    // element is "hello", "doctor", "name", etc.
}
```

INDEXING BY RANGE

Often we need to refer to a set of elements in an Array. Just like in for loops we can use range syntax (e.g. 0…3), we can do the same for Array indices:

```
// Gets the first 4 elements of the Array as a new Array.
words[0...3]
```

MANIPULATING ARRAYS

Before we start manipulating arrays, we have to understand how they interact with *variables* and *constants*.

We know that when we declare a constant, we can't change the value that constant contains:

e.g. `let myAge = 21` – myAge can't ever contain a different value.

MANIPULATING ARRAYS

Constants are intended to declare "immutable" values. That is, values that *cannot be mutated or changed.*

This applies to Arrays as well, but not only that a constant isn't allowed to refer to a different Array, but also that the contained Array itself can't be changed.

```
let shuttles = ["Columbia", "Discovery", "Challenger", "Endeavor", "Atlantis"]
shuttles.append("Enterprise")   // not allowed!

var probes = ["Voyager", "Cassini-Huygens", "Curiosity", "Galileo", "Juno"]
probes.append("New Horizons")   // okay!
```

MANIPULATING ARRAYS

Change an element in an Array bound to a variable using setter syntax on the indexed element:

```
probes[0] = "Voyager 1"
```

Add an element to an Array (bound to a variable) with .append:

```
probes.append("Voyager 2")
```

Insert or remove an element wherever you want, as long as the index is in the Array's "bounds":

```
probes.insert("Viking", atIndex:0)

probes.removeAtIndex(0)
```

MULTIPLE ARRAYS

The + operator represents concatenating two Arrays, which, like Strings, produces a new Array with the elements from both in order. The Arrays must be of the same type.

```
probes + ["Kepler", "Mariner", "Magellan"]
```

ARRAY PROPERTIES AND METHODS

We've already seen some methods we can use on Arrays to manipulate them (e.g. .append). There are other methods and some properties we should know about to work with Arrays. Here are just a few:

Get the number of elements contained by an Array using its .count property:

```
words.count
```

Get the first and last elements:

```
words.first
words.last
```

Return whether it's empty:

```
words.isEmpty
```

ARRAY PROPERTIES AND METHODS

Some important methods:

Return whether an Array contains a given element:

```
shuttles.contains("Columbia")    // true
```

Combine all the elements into a single String.

```
shuttles.joinWithSeparator(", ")    // "Atlantis, Columbia, Endeavor, ..."
```

MANIPULATING ARRAY ORDER

Sorting an Array means to reorder its elements according to some rule, like alphabetize.

Sort an array according to its natural order. Works automatically for basic types like Strings:

```
let sortedShuttles = shuttles.sort()
```

Or change the order of a mutable Array (i.e. contained by a variable):

```
probes.sortInPlace()
```

MANIPULATING ARRAY ORDER

If you have an Array of a class you've written, you can sort it by providing a "comparator" function. You can also use comparators to provide a more complex sorting rule for any data type:

```
class Dog {
    var name : String
    init(name:String) {
        self.name = name
    }
}

let dogs = [Dog(name:"Toshi"), Dog(name:"Layla"), Dog(name:"Neeko")]

func sortByName(a:Dog, b:Dog) -> Bool {
    return a.name < b.name
}

dogs.sort(sortByName)
```

A comparator takes two values and expresses which is first in order by returning a Bool. It typically reduces the comparison to basic data types.

MANIPULATING ARRAY ORDER

You can also reverse the order of an Array, but it has an extra step:

```
let sortedShuttles = shuttles.sort()
let reversedShuttles = sortedShuttles.reverse()
Array(reversedShuttles)
```

With the .reverse() method, it returns a special object that enables us to iterate over the Array's elements in reverse order without actually having to create a new Array. This avoids duplicating data.

Because of that "convenience," we have to explicitly instantiate a new Array given this special value.

MANIPULATING ARRAY CONTENT

You can perform transformative operations on an entire Array using the .map() method:

```
func getName(dog:Dog) -> String {
    return dog.name
}
let dogNames = dogs.map(getName)
```

MANIPULATING ARRAY CONTENT

You can pull out elements of an Array that are relevant for the given context. We call this "filtering," and you can do by giving a function that accepts an element and returns a Bool, expressing whether the element should remain in the Array.

```
// Assuming Dog has a .gender property containing
// an enum Gender with cases .Male and .Female
func isMale(dog:Dog) -> Bool {
    return dog.gender == .Male
}
let maleDogs = dogs.filter(isMale)
```

ARRAY DEMO & EXERCISE

ARRAY EXERCISE

Let's create a data model for managing a list of Notes. We'll put it into an app later.

- Create a class called Note.
    - Properties: title (String), content (String), dateCreated (NSDate)
- Create a class called NoteCollection.
    - It should hold a collection of Notes (perhaps via an Array property called “notes”?)
    - Create a new Note given a title.
    - Method to return the Notes sorted by dateCreated.
    - Method to get the current number of Notes.
    - Reference a single Note currently being edited by a User.
    - Select a Note to edit by index.
- Create an instance of NoteCollection and add some Notes to it.

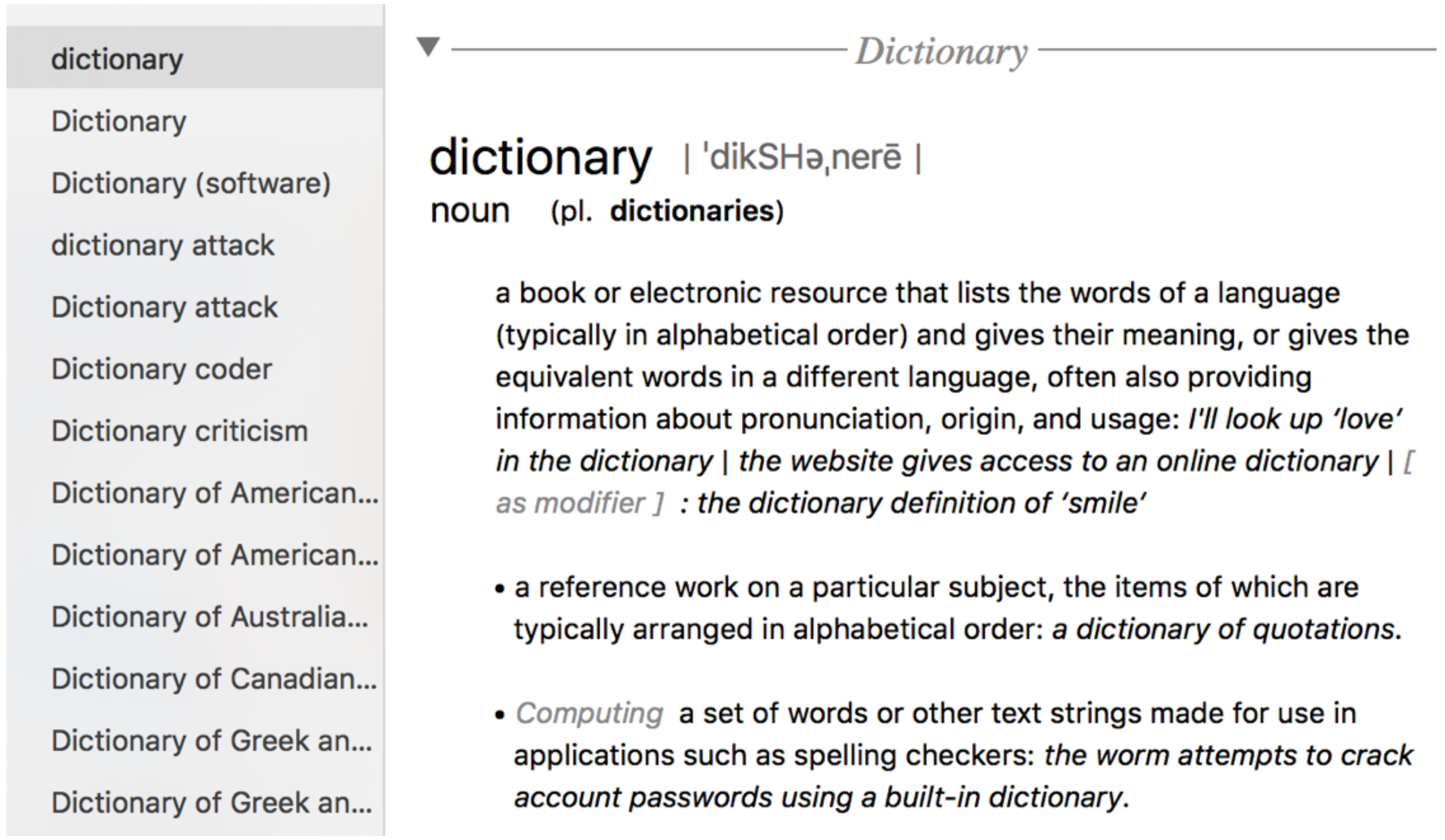
DICTIONARIES

WHAT IS A DICTIONARY?

- A Dictionary is a data structure that acts as a one-way association from one set of *unique* things (like phone numbers) to another set of things (like people or users).

WHAT IS A DICTIONARY?

- It's like a language "dictionary;" each entry (word) is unique, and each has one definition. Some words can mean the same thing.



dictionary
Dictionary
Dictionary (software)
dictionary attack
Dictionary attack
Dictionary coder
Dictionary criticism
Dictionary of American...
Dictionary of American...
Dictionary of Australia...
Dictionary of Canadian...
Dictionary of Greek an...
Dictionary of Greek an...
Dictionary

dictionary | ˈdikSHəˌnerē |
noun (pl. **dictionaries**)

a book or electronic resource that lists the words of a language (typically in alphabetical order) and gives their meaning, or gives the equivalent words in a different language, often also providing information about pronunciation, origin, and usage: *I'll look up 'love' in the dictionary* | *the website gives access to an online dictionary* | [as modifier] : *the dictionary definition of 'smile'*

- a reference work on a particular subject, the items of which are typically arranged in alphabetical order: *a dictionary of quotations*.
- *Computing* a set of words or other text strings made for use in applications such as spelling checkers: *the worm attempts to crack account passwords using a built-in dictionary*.

WHAT IS A DICTIONARY?

- It’s kind of like an array, but instead of using indices (0, 1, 2, 3, … count - 1), we are given the ability to define our own indices, called “keys.”
- *But they don’t hold order like Arrays do.*
    - i.e. We don’t know what order the keys and values will come in.

WHAT IS A DICTIONARY?

- We say that a Swift Dictionary contains "*key-value pairs*." A key always comes with a value; a value always comes with a key.
- A Dictionary has a set of unique **keys**.
  - *i.e.* "unique" = Each of key is only found once in the Dictionary.
- Each key is associated with a **value**, which can be easily referenced if you have the **key.**
  - Values need not be unique in the Dictionary.
- The type of a Dictionary is the type of the keys + the type of the values.

HOW DO YOU DECLARE A DICTIONARY TYPE?

This declares a "Dictionary from String to Double" *implicitly*. The keys are of type String; values are of type Double. A single key-value pair is written as two, colon-separated values. The key is on the left, the value on the right. Multiple key-value pairs are separated by commas:

```
var constants = ["e": 2.71828]
```

```
var constants = ["e": 2.71828, "pi": 3.14159]
```

Dictionary types use a syntax similar to Arrays, but they declare two types instead of one. Use a colon to separate them. Here is an empty Dictionary of the same type as above, declared *explicitly*.

```
var constants : [String: Double] = [:]
```

You can also instantiate the type directly:

```
var constants = [String: Double]()
```

HOW DO YOU DECLARE A DICTIONARY TYPE?

Consider a Dictionary of mathematical and physical constants:

```
var constants = ["e": 2.71828, "pi": 3.14159, "phi":1.618, "tau":6.283]
```

Note that Swift does not maintain your order of key-value pairs. The playground will display:

```
["phi": 1.618, "e": 2.71828, "pi": 3.14159, "tau": 6.283]
```

But this order is *stable*. So if you don't add/remove key-value pairs, the order should remain the same.

DATA STRUCTURES

USING DICTIONARIES

```
var constants = ["e": 2.71828, "pi": 3.14159, "phi":1.618, "tau":6.283]
```

Retrieve a value by using the key and subscript notation. Say "constants sub pi":

```
constants["pi"]
```

If you give a Dictionary a key it doesn't have, you get nil!

```
constants["G"]  // nil
```

MANIPULATING DICTIONARIES

```
var constants = ["e": 2.71828, "pi": 3.14159, "phi":1.618, "tau":6.283]
```

Add a key-value pair by using setter syntax. Use the same syntax to change the value for the given key.

```
constants["c"] = 299_800_000.0  // Creates a new key-value pair.
constants["c"] = 299_792_458.0  // Edits that pair with a more accurate value.
```

Remove a key-value pair using the Dictionary method removeValueForKey. Remember, only for *mutable* Dictionaries.

```
constants.removeValueForKey("tau")
```

ITERATING OVER DICTIONARIES

You can iterate over the key-value pairs in the Dictionary using a for-in loop and a tuple to "unpack" both the keys and the values.

```
for (symbol, value) in constants {
    // symbol will be "e", "pi", etc.
    // value will be 2.71828, 3.14159, etc.
}
```

SYNTAX: DEALING WITH ABSENT KEYS

Since accessing a Dictionary with a key it doesn't have results in nil, we can use a familiar if-let syntax to account for this case.

```
if let G = constants["G"] {
    // Does this look familiar?
} else {


}
```

MORE ABOUT DICTIONARIES

- We use Dictionaries when:
  - there is a strong association between
    - a set of *unique* identifying keys, and
    - another set of values to be identified (not necessarily unique),
  - and it makes more sense to work with those keys than an Integer index.
- Examples:
  - SSN → person
  - license plate number → Car
  - registration number → dog
- You *really should* check a Dictionary with if-let every time you index into it. Don’t assume it has the key or a value with that key!

DICTIONARY DEMO & EXERCISE

DICTIONARY EXERCISE

Let's create a data model for managing a list of Dogs. We'll put it into an app later.

- Create a class called Dog.
  - With properties name, gender (use an enum), breed, registration number, etc.
- Create a class called DogCollection.
  - Hold a collection of Dogs (perhaps via an Dictionary property called, “dogs”?), keyed off of the registration number.
  - Method to add a single Dog to the collection.
  - Method to return a list of Dogs sorted by name.
  - Method to compute current number of Dogs in the collection.
  - Return a Dog given its registration number. (What happens if no Dog has that name?)
- Create an instance myFamilysDogs and add some dogs to it.